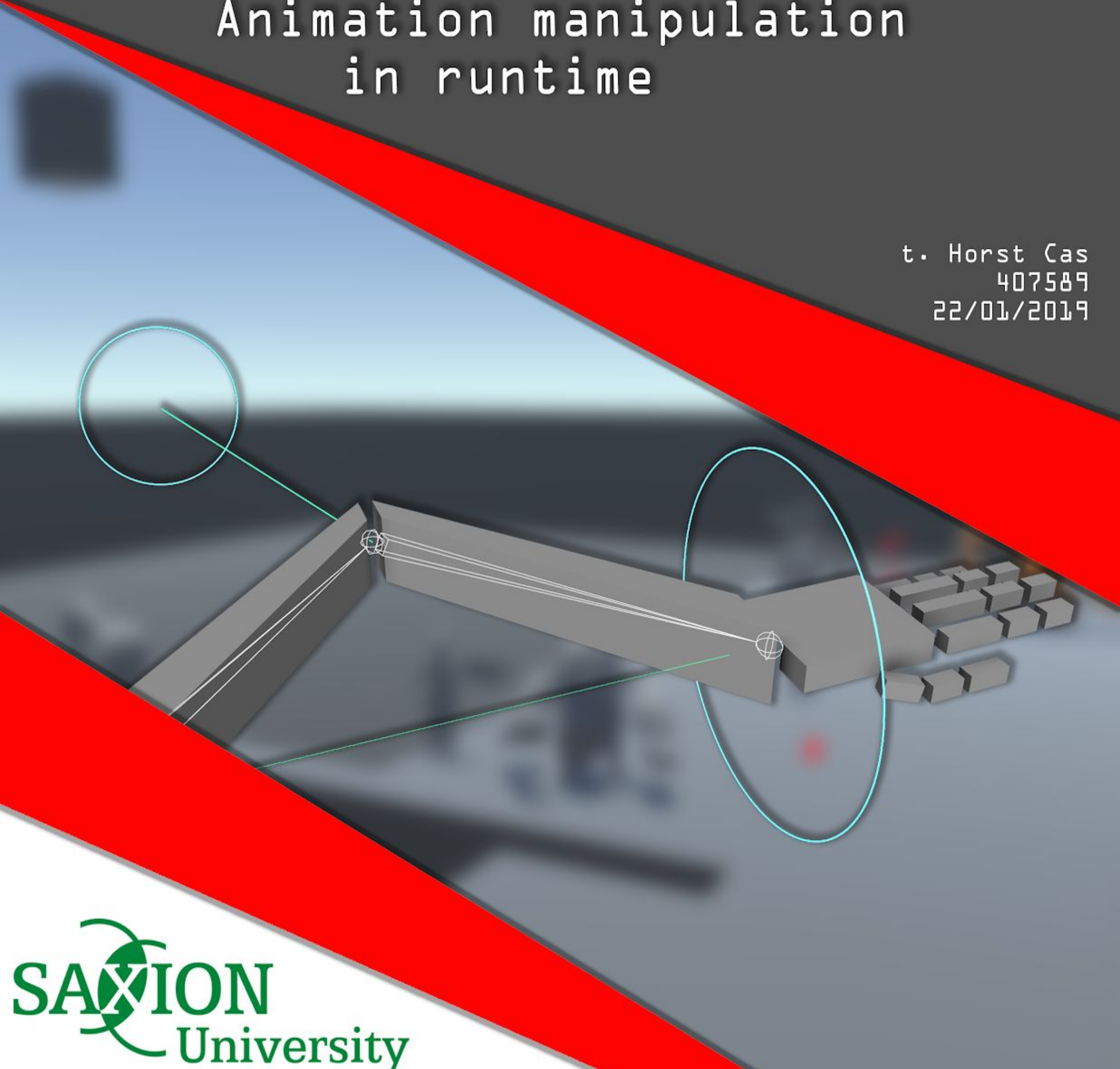
INVERSE KINEMATICS

Animation manipulation in runtime

t. Horst Cas
407589
22/01/2019



Saxion University of Applied Sciences

twinsense 360° virtual & augmented reality

Graduation

Tabletop RPG with AR & VR elements

Cas J. ter Horst

University of Applied Science Saxion

M.H. Tromplaan 28

7513 AB Enschede.

Twinsense Online Brand Excitement

Ripperdastraat 1

7511 JP Enschede.

Author Note

Cas ter Horst, student at the Academy of Creative Technology, University of Applied Science, Saxion Enschede.

This document has been written for the course: Graduation Report, meant to showcase the progress and results of the student's research and experimentations throughout the weeks, prior to the writing of this report.

Correspondence regarding this paper should be addressed to Cas ter Horst, student of the Saxion University of Applied Science. Email: 407589@student.saxion.com.

Abstract

This report is a final draft of the graduation report. It is a continuation of the implementation plan report, with added research and acquired results. It will mention the reasons for choosing the graduation assignment. The assignment being: The manipulation of animations during run-time, revolving around the IK-system in Unity, and explain the reasoning behind it. It will mention the client's wishes and the limitations and conditions that have been faced during the project. It will also depict the preliminary research done beforehand, containing research about earlier literature, documents, websites and personal experimentations leading up to the implementation report.

As a continuance, it will also contain the research, results and experimentations succeeding said implementation plan report combined with an explanation of the 12 competences and how these are imbedded in the work.

Keywords: research, animation manipulation, IK-systems, experimentations, 12 competences

Preface

It has been a few months since the start of the project. Thus far it has already been a joyous journey and an enjoyable experience. I have been immersing myself into my work. As a result I haven't always kept my journals and documentations up to date. Luckily, I have been posting on my blog whenever there is any relevant progress to be shown. However, I'd would advise my future-self to record any and all failures of experiments for the next time I am asked to defend my research, to show my development properly. I scarcely have any non-theoretical work to show, but I have a test scene set up with tons of experimentations.

First off, I would like to thank my graduation coach, Taco van Loon, for helping me get this position in this project. I had some trouble finding a graduation spot at the beginning of the semester. But with the guiding hand of Taco, I managed to secure myself a spot at Twinsense. He showed me my options and handed me some contacts for possible positions. Taco is very willing to help with anything he can and likes to make sure all my concerns are dealt with.

Speaking of Twinsense, I would like to thank Albert Hoekstra, my company supervisor for allowing me to work at his company. Even though, I do not physically work at the Twinsense headquarters. He agreed to take me in when I was down and out and in need of a graduation position. He's been a very friendly coach and has always shown interest in my work.

I would also like to thank Alejandro Moreno Celleri for his weekly visits and advise. Even though he is not my guiding teacher, he is always open to answer questions and seems to be genuinely interested and concerned about the happenings within the project group. I haven't always had the pleasure of meeting with him on a weekly basis, due to the sheer size of our project team, but his messages and concerns were always relayed by the team leaders.

Furthermore, I would sincerely like to thank all my teachers which have guided me thus far, throughout my years here at Saxion for all the work they put in me. Every single one of them seems to love their work and have an honest interested when it comes to helping the students. They've put up with my nagging and stress-filled emails, when I believed I was running out of time for assignments submissions and the like.

I would also like to thank the team members of my graduation project group and any project group I was a part of in the past for that matter. My team helped me out when I got stuck and needed some insight from a different perspective. Especially when the programming got though. I only needed to ask and they were ready to help me understand and help resolve my problems.

Lastly, I would like to thank anyone who I still have forgotten to mention. I appreciate any and all things you have done for me. I am proud of myself for making it this far. I couldn't have done it by myself.

Table of Contents

3. General description

A recurring problem when playing video games, is that the immersion can be broken quite easily by something unexpected or unrealistic happening. One of these problems is the player character acting abnormally, when interacting with the environment. The character is out of place or doesn't quit react in a realistic manner, if it reacts at all. When this happens, the player snaps back to reality and loses focus of the game, which goes hand in hand with his or hers enjoyment.

"Immersion is a metaphorical term derived from the physical experience of being submerged in water." (J.H.Murray)

Most games use animations that are quite linear. For example: walking, running, jumping or swinging a sword. These animations can be visually pleasing by applying physics and other neat tricks, such as blending sideways walking animations for smoother transitions. Basically, having more senses assaulted with sensory information at once, the better or stronger the immersion will be. (*J. M. (2015, October 20). The Psychology of Immersion in Video Games*)

Bad animations can have a negative effect. For example, the latest Mass Effect Andromeda, developed by BioWare and published by EA. Player reviews suggest that players had a hard time being immersed and joy the game. One of the reasons being due to the facial animations. These small, but significant details, can set off a trigger in your brain, bringing you out of your immersion and possibly ruining your enjoyment of the game. (*Stapleton, D. (2017, March 20). Mass Effect: Andromeda Review*)

A different example of breaking immersion that fits better with this research: When a sword doesn't quite reach its target in the correct manner or angle, ór when a foot slides or hovers above the ground. The player will notices all these small details subconsciously and when they all add up, the player will again have a hard time being immersed. Luckily, animations, such as a sword swing, can be edited in real time via IK controls within the engine, to a certain degree. This allows the animator to set targets where the sword can hit or its foot will land. This will create a more believable animations and keep the immersion going.

Using IK-systems, also allows for reusability from the same animations, while holding different weapons and with different targets, thus optimizing the workflow and efficiency. The animator doesn't need to recreate all animations to fit every character, except for when the body is too differently shaped. Alexander Bereznyak talks about IK and gives demonstrations at the GDC. (*Bereznyak, A. (2018, January 19). IK Rig: Procedural Pose Animation (GDC, Ed.)*)

4. Client's Objective

According to the client's wishes, augmented reality and virtual reality features need to be implemented into a 3D tabletop-like game. Preferably with the requests, and vision of the client implemented. The clients requires the students, which are assigned by the Saxion University of Applied Science, to create a sociable and cooperative game, that thrives on immersion and storytelling. It should be viable enough to showcase the possibilities between augmented reality and virtual reality in a fun manner, and allow for future projects.

.

5. Limiting conditions

When working with augmented and virtual reality, there are a plenty of limitations that need to be taken in account. Especially when combining the two in a single system. For example, VR renders every object in the scene twice, because it uses double lenses inside the headset. It also needs to have a positive 90 frame rate, otherwise players can get nauseous. Luckily VR is usually played on a desktop, thus the power it can handle is significantly higher than an AR game run on a mobile device. (*Martindale, J. (2018, December 17). Oculus Rift vs. HTC Vive: Prices drop, but our favorite stays the same*)

By combining VR and AR, it is possible to let the VR player host the game on their desktop PC and let most of the calculations be done by the PC, instead of the mobile phone. This enables more power to be utilized by the phone for other calculations. The PC must of course be capable to handle those calculations and needs a certain bandwidth to establish a stable connection to the server.

6. Main questions

To refresh the memory, the research topic is: How to manipulate animations in runtime, by use of IK-systems in Unity. The biggest challenge is creating animations, which won't break the immersion of the player. The plan is to help create better looking animations to keep the player immersed and focused. It would be a shame, if the results of the research, are the reasons that caused the immersion to break in the first place. Fortunately, there are a few things available online about this topic that can help with figuring out a solution to the problem. But sadly, most of those are not very detailed and some are only intended for a different engine.

To answer the main question, several sub questions need to be answered as well. Some of these questions can be hypothesized beforehand, but some new question will certainly arise during the experimentations in Unity.

6.1. Sub questions

The several sub questions that have come up beforehand will help to answer the main question. There a multiple things that need to considered and researched. One of those things that needs to research, before animating can even begin, is the need to understand how the basics of IK work and what the best possible rig is. What could be wise to figure out as well, is how to attach weapons and other props to the character. The weapon would of course be the object that needs to reach the target and not the hand that holds it.

The sub questions that have arisen are broad and might even consist of multiple sub questions themselves:

- How does inverse kinematics work within Unity?
    - Does it function differently than previous experiences?
- What are the limitations of Unity's IK system?
- How can the position of the effector be decided?
- How can the targets change seamlessly?
- What type of rig is consistent with IK?
- How does inverse kinematics work for characters with a different rig?
- How can an event system be attached to work harmoniously with the IK system?
- What things must absolutely be avoid to make it work properly?

These questions will be answered under the "*The approach*" headline, but a separate journal will be added to help get a better understanding of my thought process and experiences.

7. Theory

7.1. Known research

The Unreal engine already has inverse kinematics implemented in their engine for a few years now, under the name "*Ikinama*". However, Unity3D has not. The plan is to do research on how to get this to work properly in the Unity engine, to a certain degree. The Unity engine is used for the current project after all. Unreal has made their programming system quite easy to understand via their Blueprint system, which allows people who do not know how to code, to be able to achieve the same results, as someone who does know how to code. The Blueprint system is essentially programming with nodes, each node has a certain code or function connected to it. *(Engine, U. (n.d.). Blueprint Best Practices)*

It comes with its limitations however. If the user want to create something unusual or entirely new, the system can't make it work, or can only make it work to a certain degree. Unity gives the user a lot more freedom, when it comes to what can be done within their engine. But it is also quite a bit more difficult for an inadequate programmer, who isn't familiar with C#, which is the programming language Unity uses for their scripts. *(Documentation, Unity scripting languages and you – Unity Blog. (n.d.))*

The way this works in Unreal is pretty simple to understand, thanks to the documentation on their site and their live tutorials on YouTube, hosted by the Unreal staff or experts. Of course the engine works differently than Unity's, but having a basic understanding of how this works can help me figure out how to get it to work in Unity. After all, there might be some similarities that can used. (*IKinema RunTime and Unreal® Engine 4. (n.d.)*)

Another great example, are the popular Assassins Creed games. The climbing system in the series uses the Inverse Kinematic system prominently, since scaling buildings is a key feature of the games. Ledge grabbing via normals and edge detection, allows for the character to correctly place its hand on top of a ledge, via IK. Michael Milord talks about the system in one of his videos, explaining the limitations, constraints and possibilities. (*Milord, M. (2016, January 11). Developer View: Assassin Creed Unity Parkour 1*)

Regarding how this could work in Unity, there is documentation on their site that explains the basic workings of IK in the engine. The built-in functions that can be called upon in the scripts make the code a lot easier to handle, but there are tons a variables that come into play when writing the code. Combined with forums found on google, of people who have tried to approach this topic as well, and have ran into problems. There is plenty information to be found and to be researched. There also are a few YouTubers out there that post tutorials on how this system works in the engine and explain the possibilities of the system. (*Technologies, U. (n.d.). Inverse Kinematics*)

7.2. Known problems

The known problems about using IK-systems in Unity, is that it possibly requires more power due to the runtime adjustments of the bones, the GPU has to do more calculations. But the difference between the risk and the results is neglectable. The pros far outweigh the cons, especially since we are having most calculations done by the host PC.

IK works by transposing the bones in the chain via an algorithm. The first bone in the chain gets overlooked, because it's the effector: The bone that needs to reach the target. The second bone transforms first and moves the effector closer to its target. Then the 3rd bone transforms, then the 2nd bone again, then the 3rd and 4th bone and so on, until the effector has reached his target or has gotten as close to it as possible. Via this Method, the entire joint chain will evidently transpose incrementally, but the first few joints in the chain will transpose the most. The further down the chain, the lesser the transformations.
Lukas Barinka and Roman Berka talk about such methods in their paper and describe how they work. (*L. B., & R. B. (n.d.). Inverse Kinematics - Basic Methods[PDF]. Prague: Dept. of Computer Science & Engineering*)

Luis Bermudez explains the rough workings and differences of Forward Kinematics and Inverse Kinematics, but in a way that is much easier to understand.
(*Bermudez, L. (2017, July 10). Overview of Inverse Kinematics – Unity3DAnimation – Medium*)

Another issue can be that the model can deform in an unrealistic manner, especially if the model is an organic character. Therefore, certain constraints need to be set for when using this system. If a target is too far away for the character to reach it by simply stretching its arms, then the spine can be transformed as well, to help extend its reach. If the effector still cannot reach, then the effector will simply get as close to the target as possible. However, this might cause to model to deforming into an unrealistic pose, where the arm is completely stretched and the spine bend forward. Unity’s muscle system, that can be accessed when configuring the avatar, can limit how far a humanoid character can bend.

8. Problem definition

The problem the client has brought up, is a proof of concept where augmented and virtual reality are combined into one demo. The ultimate goal of the client, is of course to generate revenue with this concept, where the client's wishes regarding AR and VR are met. By demonstrating the game as a demo or prototype, the client will be able to gain some media attention and in turn, be able to appeal to a new branch of organizations that could require Twinsense's services.

However, the research of immersion through manipulation of animations in runtime, is only a small part of the ultimate goal. The research and experimentations alone will not be able to satisfy the client wholly or even satisfy the clients needs at all. The client is not in the gaming business and would possibly have no need for the provided research, but by working with the other students, the team can complete the client's request and make sure the client is indeed satisfied. The client is mostly interested in the technical part behind the project, but requires the beautiful aesthetics to help sell the idea.

The original research question was supposed to involve manipulation of animations by means of physics. However, having too many physics dependable objects in the game will extremely increase the processing power required to run the game, and the game will not be as fast paced where one might notice these physics in play. Thus, a decision was made to take a different approach on animation manipulation, by swapping out the physics and implementing the IK-system instead.

IK-system based animations are a little more predictable and controllable and can drastically change the workflow, when factoring in the reusability of the same animations for different movements. For example, by adjusting the position of the hands, a character can hold a variety of weaponry or can change the way he walks and moves, in such a way, that the original animation is barely recognizable.

9. Scope

The limits of the assignment are pretty clear. The focus lies on animations. That includes the subtasks like rigging and weightpainting as well. Those tasks are essential to creating visually pleasing animations, but are not necessary to finish the research. Those are tasks that have to be done in order to achieve the goals in particular fashion. A simplistic dummy can be used to showcase the result, but having an eccentric character follow the movements draws more attention.

Nevertheless, there will most likely have to be done some minor 3D modeling and texturing to help with achieving the goals. For example, if the plan is to be able to cut a character model limb from limb, destructible character will need to be created. But that will be the last step, which will only happen when all the other things turn out to be done sooner than expected.

As estimated, a lot of time will be spent on writing code for the IK-system. Unity does not work with visual scripting like Unreal engine does, thus a concrete understanding of the basis of the programming language, which is c#, is needed.

Besides models, textures and animations, there is a lot more when it comes to developing a game. The other topics can be summed up as, game design and visual effects. These tasks will not be focused on deliberately, simply because there is no desire to gain any more knowledge about these topics and it will not help with making natural looking animations.

10. The approach

10.1. Methods

To answer my main question: How to manipulate animations in runtime by use of IK-systems in Unity, a playable prototype will be constructed. In addition, a showcase and a possible gag reel of bloopers and failures, that have been encountered throughout the process, will be added.

.

The plan is to create a separate side project in Unity, where all experiments with the IK-system and all of its settings will be done. Multiple animations, rigs and IK-systems will be created to help figure out the optimal usage of the system. All the ins and outs on how the system functions can be seen and notes taken of its limitations and conditions.

To answer the sub-questions, research and experimentations is required. This will be done by create a test scene, where multiple animations and models can be imported and set up with the IK-system. The animations will be played on the custom models, created by the project team and then be used to figure out how the IK-system overrides the current playing animation. The next thing that needs to be understood, is how does the effector and target correlate; if multiple targets can be set and how to switch seamlessly between them; and if the strengths of the override can be blended to create seamless transitions.

11. Answering the sub questions

11.1. Differences

The first sub question that needs answering is: How does inverse kinematics work within Unity and does it function differently than previous experiences. Inverse kinematics is widely used in a variety of 3D software to create animations, but what was still unfamiliar, was that it is also used in game engines to adjust animations.

A quick summary to explain the basics of inverse kinematics: By translating the effector, the last node in a joint chain, the other joints in the hierarchy will move accordingly to allow for the effector to reach a certain target. Basically meaning that, by moving the wrist, the elbow bends and the shoulder rotates to allow for the hand to be placed against a surface.

The big difference between inverse kinematics in a 3D program like Autodesk Maya and Unity, is that the translations of the bones, caused by pulling on the inverse kinematics effector, is baked onto the joints before exporting it to the game engine. Which means the IK gets removed from the skeleton and is only used to determine the pose or the movements for the animation.

In Unity, the IK system is running in real time. Which means that there can always be an influence from a target to adjust the animations. The computer normally only has to calculated the translations of the joints from the animation itself, but now there is an external factor that can change those rotations on a whim. Causing the computer to do more calculations to allow for that change to happen.

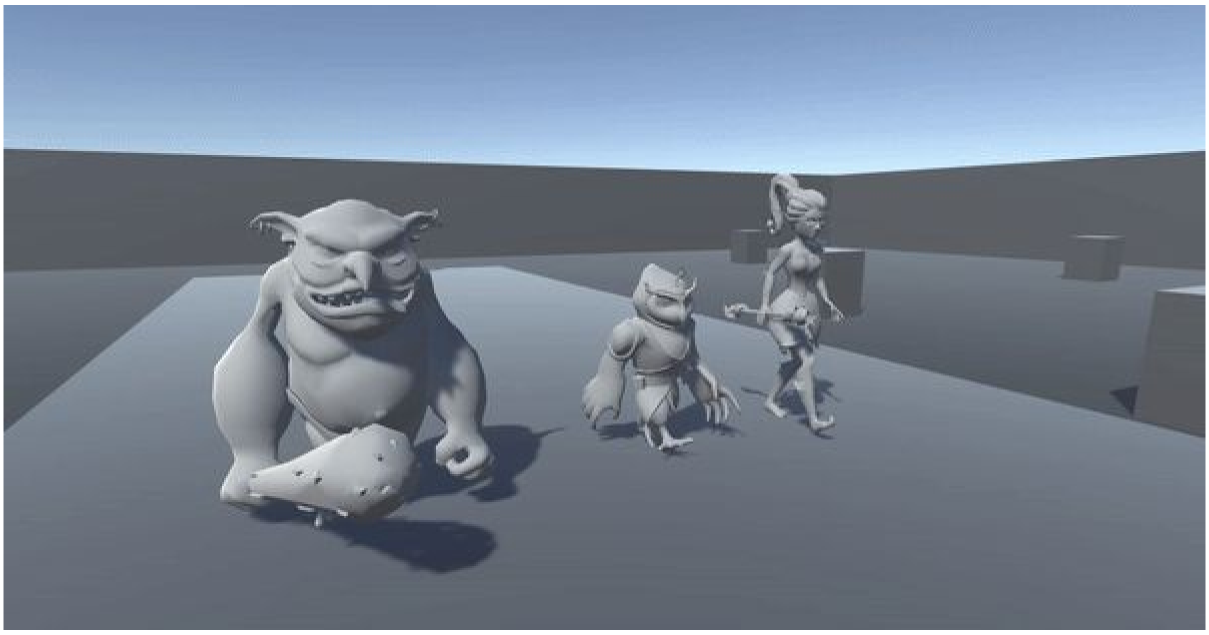
11.2. Workings of IK

The way inverse kinematics works in Unity is fairly simple to describe, but harder to accomplish, because IK works via built-in functions in a C# script. A skilled programmer, who is experienced with the language, might not have too much difficulty using the built-in functions to all of it's extends, but the functions can be used in multiple different methods and for different parts of the body. Such as the arms, legs, spine and neck.

As described on the Unity documentation page (*Technologies, U. (n.d.). Inverse Kinematics*), inverse kinematics can be used to grab and hold onto objects, such as weapons. This works by setting targets where the effector can be pulled towards too. But it can also be used to adjust animations in a more subtle way.

For example, if a character needs to change the way he/she walks, or sway one of its arms a bit less. It is possible to set a target close to the body itself, with a low influence, that will ever so slightly pull on the effector to differentiate from the original animations. Causing the character to walk with an arched back, have a limp, have one of its hands in its pocket or even have a broken neck. (*AiGameDev.com. (2015, June 10). Nucl.ai Conference: Ubisoft Toronto "IK Rig" Prototype*)

The image on the next page is a gif of one of our custom characters (left), which receives animation overrides to create a better fitting walking animation. Find the full URL for the blog post in the Annex under the headline "*Changeable Walk*".
There will be more blog URLs linked in this paper. For convenience purposes, keep a separate tab open with the blog. View the blog here: https://tabletoprpgblog.wordpress.com

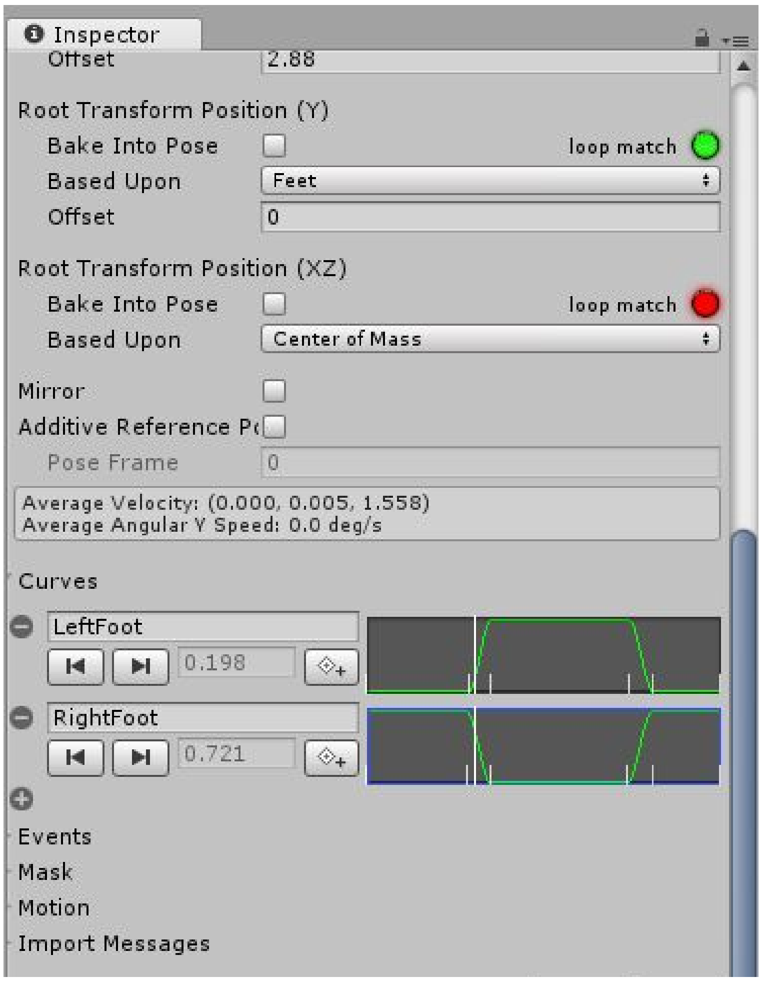


Lastly, a good example that is widely used in games, is inverse kinematics for the feet. Meaning, that the feet will always be pulled down onto the ground when walking on uneven terrain or even stairs. This is done by casting a ray downwards, from the position of the foot, until it hits the collision box of a ledge. The raycast would then translate the foot, to the point where the ray had hit, and rotate the foot according to the normal of the surface.

The code required to make all of this happen is in its essence, very basic: Call upon the animator, set up a target and create a new void function called OnAnimatorIK, wherein the majority of the code will be written. This function allows the influences of the target to override the current animation clip that is being played. One very important step that has to be done, is the function requires the IK Pass to be enabled in the animator or all the code will be ignored.

Inside the function, the target can be set and the amount of influence can be determined. The target is the only thing that needs to exert influence on the effector, ranging from 0 to 1. 0 being no influence and 1 being the max amount. It is also possible to determine if, both the translation and rotations of the target is needed to influence the effector, or just one of them.

The value of the influence is a float, so the transition from the original animation pose, to a full override can happen smoothly. This can be done overtime or by adding a parameter with a curve. The curve is especially handy when working with Foot IK to determine when the foot is placed down on the ground or lifts up.



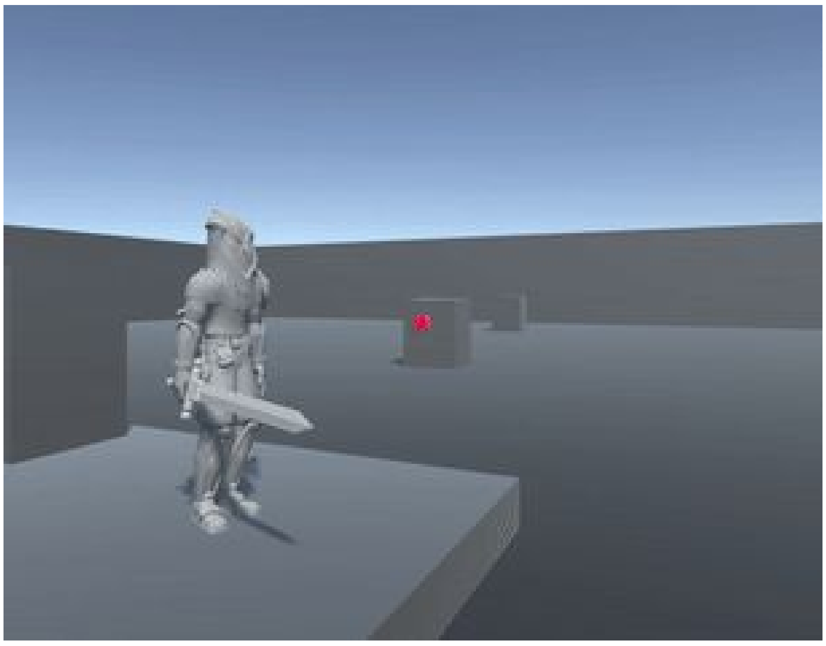
```
Inspector
Offset                          2.88
Root Transform Position (Y)
  Bake Into Pose      ☐                    loop match ●
  Based Upon          Feet
  Offset              0
Root Transform Position (XZ)
  Bake Into Pose      ☐                    loop match ●
  Based Upon          Center of Mass
Mirror                ☐
Additive Reference P( ☐
  Pose Frame          0
Average Velocity: (0.000, 0.005, 1.558)
Average Angular Y Speed: 0.0 deg/s
Curves
  LeftFoot    0.198
  RightFoot   0.721
Events
Mask
Motion
Import Messages
```

The Annex contains two script: "*Foot IK*" and "*Foot IK Pro*", which use the attributes of inverse kinematics for the feet. It also contains two downloaded scripts: "*Reference Foot IK 1*" & "*Reference Foot IK 2*". They were used as reference to better help understand the workings. They have been slightly altered due to experimentations with the scripts.

11.3. Limitations of IK

Inverse kinematics sounds incredibly versatile, but there are of course a few things it cannot do, or are very hard to do. Speaking from previous experiences when working with IK. As observed, inverse kinematics has a hard time adjusting animations that are not grabbing objects or placing the feet on the ground.

Animations, such as swinging a sword into a target, seems very difficult to accomplish. The main cause of this, is the target being too far away. If the desired result is for the blade of the sword to slice into the target, which is actually pulling on the effector on the wrist, some mediocre results are achieved. Since the target is too far away for the hand to reach, the arm will stretch to its fullest extend and it influence will cause to effector to linger at the targets position too long, making it look like the swinging motion becomes a stabbing motion.



The URL to the corresponding blog post in the Annex under the "*Slashing*" headline.

Another limitation is the rotations that can be copied from the target. The target has to be set up in such a way, that the rotations of the effector will not be twisted in the wrong axis. This might not be too difficult to achieve for a handful of preset targets, but it does require a solution for when there are dozens of targets which could change in runtime.

The apparent solution was to add a script to each target, which always looks at the effector. The base rotation of the target is wrong it the first place, and the LookAt function, that is used to allow this to work, disregards the original rotation given to it. Thus the target is given a rotation offset in the script. Which allows for a sword to slice the target from the below, the side or from up top.

The last limitation: inverse kinematics currently only works on humanoid rigs and does not work on generic rig. It is being worked on to make it work on all types of rigs, according to Unity's roadmap.

11.4. Effector

One thing that really needed the use of inverse kinematics, was changing combat animations, such as swinging a sword or throwing right hook punches. But while messing around in Unity, it is seemed that the effector of the hand was being pulled towards the target, which almost looked as if the sword was slicing the target. Reflecting on the findings, it became apparent that the effector had to be the last node in the joint chain, which in this case was the hand joint, but the question that remained was whether or not it could be an empty game object in the hierarchy, which could be placed in the blade of the sword.

The way the built-in functions works, is by calling upon the "*AvatarIKGoal*". Which is always one of the hands of feet. It doesn't allow for simply putting in a different gameobject in its slot and expecting the function to work. But what can be done, is giving the IK effector an offset, or better said: the effector can stay in place. When working on the feet IK, this came to light. Unfortunately, recreating it, to try it out for the sword swinging animation set up in Unity has been futile. With some more time to work on it, there will be another attempt to try and see if it is indeed possible to recreate the code and place an empty game object in the blade of the sword. Making the empty game object the new position of the effector.

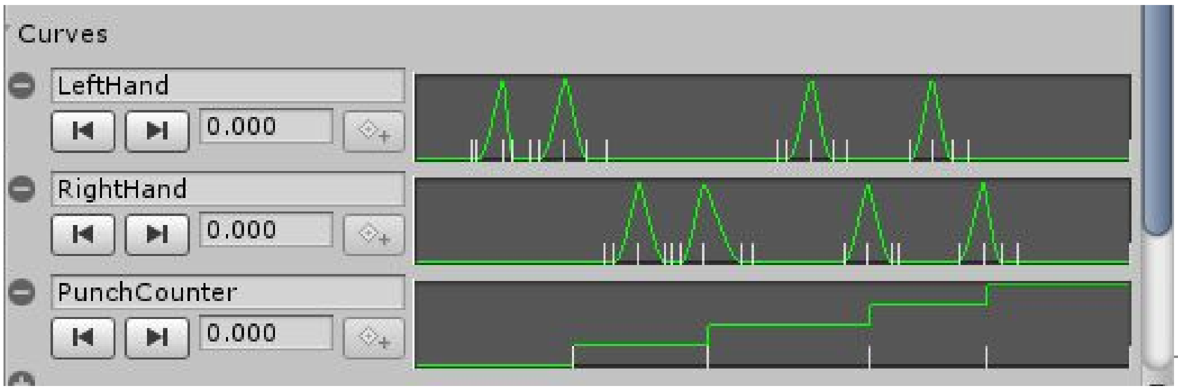
11.5. Changeable targets

One major thing that needs to be achieved, is to randomize the influence of the target in runtime or decide via set parameters which target should exert influence. With parameters, the engine can decide where an attack should hit depending on the critical strike factor and luck. Making sure the experience stays surprising and fresh. The hardest part about changing targets, is of course the coding, but besides that, it is difficult to determine when a strike or a punch has landed. The thing that needs to be taken in account when changing between targets, is that the transition needs to happen incrementally, so not to cause any snapping.

The coding seemed more complex than anticipated. In essence, the code can hold an array of targets that will be randomly picked, or could in essence be picked depending on set parameters. Then, depending on the animation, a curve needs to be added to tell the parameter when a hit has occurred. This could possibly be done by adding colliders and hitmarkers to the targets and the weapons. That would require more setting up, but could ultimately give better results, due to not being animation dependent, but physics dependent.

An early iteration of the script can be found in the Annex under the headline "*Punch Combo*".

The image below is of the curves used to determine the influence of the inverse kinematics and the curve used to determine how many punches have been thrown.



11.6. Skeletal compatibility

As mentioned before, the humanoid rig is currently the only type of rig that is compatible with the inverse kinematics system in Unity. But for a humanoid rig to work properly, the avatar needs to be set up correctly, so that the AvatarIKGoal, which is called upon in the script, matches with the correlating bone.

Setting up the avatar is quite simple, one can just drag-and-drop the bones in the correct slots, but that is not necessary when working with the correct naming convention. Unity will automatically detect the bones and map out the avatar. Only thing that might be required is to set up the T-pose properly, so that the animations will be retargeted correctly. Unity’s “*Enforce T-pose*” option does a decent job of getting it right, but it's usually pretty crooked.

11.7. Event system with IK

What is meant by an event system with IK, is scripting the events, so that condition will change automatically during runtime, without having to manually change the settings and values. This is especially handy when, for example, a character picks up a different weapon and needs to swing it from a different angle. An event system is immensely complex with a lot of "*if*" and "*else*" statements that determine the next action depending on the current condition. When implemented correctly, it can allow for a lot of varying situation that can keep the gameplay original and fresh.

The idea was, that characters can change their angle of attack by the randomizing the positioning and rotation of the targets throughout the enemy body. Maybe even, depending on what target is activated, play different animations, so that the same animation is not repeated with just an offset. But change the animation from a side swinging to an uppercut or even an over-the-head swing.

Unfortunately, this part hasn't been researched yet and it is doubtful it will ever be implemented, due to the fact that the current state of the project is not ready for it, yet. Hopefully, by the time the animations and characters are done, it is possible to implement these functions and show a demo that is capable of demonstrating the inverse kinematics override.

12. Conclusion

Inverse kinematics has shown to be quite capable of a variety of different methods to adjust animations in runtime. It does however come with its limitations and flaws. A few of these limitations turn out to be just steps that need to be performed in order to get inverse kinematic to work. Some of these steps however, should already be performed when setting up a basic humanoid character within Unity.

For example, for inverse kinematics to work, the character needs to have a humanoid rig, since the other types aren't supported yet. Evidently, the avatar should be set up correctly as well, to some extent at least. It isn't necessary for playing the animations, but the movements can look off if not done right.

The IK Pass in the animator window needs to be enabled as well. It tells the engine which characters can be influenced by inverse kinematics, otherwise the function will not be able to adjust the animations like desired.

The biggest encountered limitation has to do with the amount of influence given to the IK effector and the manner in which that influence is given. In the research is shown, that the influence that pulls on the effector is controlled by a parameter curve, connected to the animation itself, to determine the placement of the effector. With further research, it is shown that the influence can be adjusted via different, much more elaborate methods, which don't necessarily give better results, because snapping can occur if the method used isn't fully worked out.

For example, inverse kinematics for the feet with the usage of curves can transition smoothly, but the lingering influence can disrupt the animation, making it seem as though the foot is glued to the floor for a moment. Contrarily, an elaborate IK methods, as seen in the annex under headline "*Reference Foot IK 1*", which determines the placement with a more mathematical approach, can look great on uneven terrain, but still snaps when the difference in height is too sudden. Therefore, it is wise to implement a prediction or a smoothing factor that can determine when a sudden height difference is close enough to step on.

Some limitations are of the users own creation and shortcomings, depending on what the IK function is used for, because Unity only provides the user with the necessary tools. If the user doesn't understand the function and the coding language fully, he/she will be limited by their own inadequacies and won't be able to fully utilize the function or explore different methods.

The IK system is basically a pull-towards-target function. Therefore, back and forth motions work excellent, but swinging movements suffer from lingering influence of the target, making it appear as if effector is stuck at the target's position, until the influence has decreased enough to continue with the animation. If this happens too sudden, the character will snap back to its original pose.

If the target has been set too far away for the effector to reach and the influence reaches the max value, the limbs of the character can unnaturally stretch to try and reach it. Which in some situations might be desired when grabbing onto an object. But when the only thing that needs to be adjusted is the angle of the arm, to keeping the animation the same, it's better to rotate the entire upper body, with a different IK function, instead of pulling on the effector.

A different method, that has nothing to do with inverse kinematics, can also be used to achieve a similar result. The LookAt function allows for rotation of the shoulder bone, in such a way, that it essentially fakes the effector in the wrist being pulled towards the target. This method is an unorthodox workaround and is not meant to be used in this way. It requires much fine tuning to get it right, but can essentially give better results in some cases.

13. Appendix

13.1. Reflection

Throughout this project I've learned an immeasurable amount of new knowledge. I do think I've answered all my research questions and I am somewhat proud to say I answered my main research question: How to manipulate animations in run time. But I feel as though I have barely accomplished anything. This is easy to say in hindsight, though. Most of the research and experimentations I have done are still in the prototyping phase and require more in-depth knowledge and expertise to make it work successfully.

I am planning to finalize as much as I can on my own by sticking to it and I will strife to have a better understanding of the programming language. I do believe that making animations isn't just setting keyframes on the right time frames, but using out of the box methods and technical methods, to improve the animations you've made. All to create more elaborate and eccentric movements, which will astound the player. To properly look back and reflect on my project, I will be writing a bit for each of the 12 competences to show my capabilities.

"Use those talents you have. You will make it. You will give joy to the world. Take this tip from nature: The woods would be a very silent place if no birds sang except those who sang best."

- Bernard Meltzer.

I. Technological competences
1. Technical research and analysis
After meeting with the company, I realised that the company is very technically oriented. They focus more on the the hardware and the programming than on the aesthetics. My goal was to deliver immersive animation that can be adjusted in runtime so that the work could be reused and optimized. I researched inverse kinematics and other similar functions in Unity and set up a demo scene to demonstrate it to the client. The client was pleased with the results and I kept working to improve the demo. I'm happy that the client liked the results, but the client probably won't have any need for the assets in future projects, since the client isn't a game developer.
2. Designing. Prototyping and realizing
During the (bi)weekly meetings, I showed the progress I had made and demonstrated new methods I found to solve certain problems. I set up multiple characters with different settings and played them side by side and steadily improved upon earlier iterations. The client could see I had been busy working and complimented me on my progress. I would've liked to make more progress, but my own shortcomings kept me from achieving more.
3. Testing and rolling out
During the design process, iterative results needed to be tested. My goal was to root out any flaws. I tested every prototype, I fine-tuned it until it worked. I created a playground that could replicate multiple situations. Most of the results turned out to have some issues or limitations, which needed some expert attention to properly solve. I delivered the demonstration to the client at the end of every meeting. I believe the client was impressed with the progress, but didn't really understand the value of it.

II. Designing competences
4. Investigating and analyzing
With the research I gathered from multiple online sources and tutorials, I had to multiple methods to experiment with. I need to play around with the settings and code to figure out which one was the optimal one. My goal was to eliminate un-useful methods or combine parts of the code. I ended up with a functional prototype that somewhat worked like desired. I needed more knowledge of the coding language to fully optimize it.
5. Conceptualizing
The client is interested in creating a VR and AR game that demonstrates the possibilities of the combination. However, my skill set is limited to producing assets. My goal therefore was to focus on the adjustment of animations to help sell the concept. I believe the client understood that my work is related to the success of the demo and liked my addition.

6. Designing
The methods are ready to be implemented, but the client would still need to have an expert programmer take a look at the functions to fully flush out the various methods. I wanted to finalize the methods more to fit the game better, but the basics are working fortunately.

III. Organisational competences
7. Enterprising attitude
The market possibilities for a game, or any other software, with both AR and VR implemented lies wide open, since very few people have attempted it. However, the target audience for my work are game developers and related designers. My work can be considered to be sold on the Asset Store, or on various 3D websites, or be used in future similar projects. I’m excited that, once the methods are fully flushed out, they can be sold online to create a small revenue.
8. Enterprising skills
The client mentioned, that the project results will be used to demonstrate the possibilities of AR and VR. Possibly to attract customers. My task was to help with creating the visual aspect. I demonstrated the possible usages of my research and results and help the client see benefit from my work. I was happy that I could help create possible revenue for the client.
9. Working in a project-based way
The project is executed with the cooperation of Saxion, with a team of students from multiple different studies. I was one of the few with knowledge of the entire process and the skills to perform certain tasks. I have unofficially worked as the quality assurer and asset optimization and the majority of the creative work has been checked by me. The client works in a different branch, but has a good eye and could see that the assets were of a good quality. I liked being a person that people came to, to check their work or ask questions.
10. Communication
During meetings with the client, I had to showcase the progress made. My goal was to make sure the client knew what I was working on. I sat down with the client and explained the progress, the workings and limitations of the current state of affairs. The client always listened to my explanations and asked question. I was happy that the client showed interest in my work.

IV. Professional competences
11. Learning ability and reflectivity
The progress often got halted by an unforeseen issue that sometimes couldn't be solved by myself. I needed to get feedback and insight from others to help better understand the problems. I requested help from colleagues and reached out to friends or teachers with expert knowledge of the topic. As a result, the issue got resolved, or worked around, within hour, sometimes days. The new knowledge could sometimes be used to fix other issues as well.
12. Responsibility
To make the project succeed, I had to focus on other work than just my own. I contributed to the project in various ways, besides doing research and doing experimentations with my code. The research I have been doing, is something I haven't focussed on before and proved difficult for me. It more technical than what I am used to. As a result, I sometimes ended up neglecting my research, because the other work is more familiar to me and I was the only one who actually knew how to do it. Plus I am better at it. I liked doing those tasks because it made me feel more part of the team.

13.2. Journal

Now that I've gone through and answered the sub-questions, I will share with you my journal of the past few months and describe how I've used the previous mentioned methods to tackle each obstacle. I will also share the other work I've done during this project in a somewhat chronological order and explain my role in the team.

13.2.1. Optimal skeleton

The first task I immersed myself in, was figuring out the rig best suited for retargeting animations onto. I had some prior knowledge when it comes to rigging, but I would say I was only adept at it at best, seeing as what I know now. The plan was to use one rigged character to function as a base or template, where all other rigs and animations could be derived from. I saw examples from Alexander Bereznyak at the Unity's GDC talk and believed I could recreate something similar, where multiple characters are driven by the same animation, but look unique in their own way.

I used the HumanIK rig from Maya, because it is generally used as the default template for characters rigs and it is compatible with all platforms. The naming convention is already set up as well, all that is required is to place the joints accordingly. I built a dummy around it just to help visualize the movements. I wanted to use this dummy as the base and began experimenting with the rig to try and figure out what would work and what wouldn't.

When researching inverse kinematics, I stumbled upon a FK/IK switch (Forward Kinematics/ Inverse Kinematics) in Maya. I always had trouble with creating smooth combat animations, due to the fact I had always relied on inverse kinematics by itself. But by adding a switch, that would allow me to swap between the two, I could, in essence, create smoother animations where the character would swing a melee weapon.

The difference between IK and FK, is that forward kinematics allows you to change one node at a time and let the other joints further down the chain stay in the same local position. By simply adjusting the shoulder area, I could create a swinging motion which would otherwise be a painstaking process to get right when using inverse kinematics. IK always tries to find the shortest route from point A to B, disregarding any circular motions.

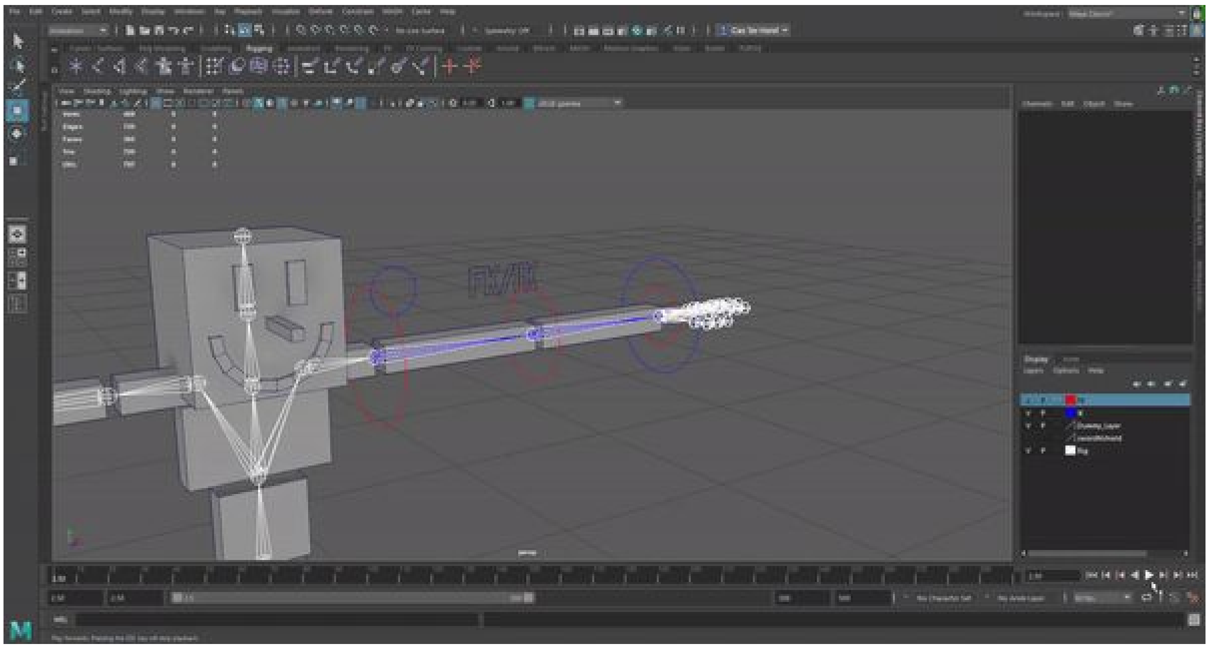
13.2.1.1. Joint setup

The way I had set up the system, is by duplicating the original joint chain that needed to be controlled by the system, twice. One duplicate will function with forward kinematics and the other with inverse kinematics. I added the applicable constraints and IK handles and once again constraint the original to the duplicates. Thus the original is driven by the other bones. Then it's only a matter of setting parameters to tell which of the duplicate bones needs to exert influence on the original bone and which one doesn't. By letting this be controller by a simple float slider, your work process becomes considerably more efficient and no snapping will occur when switching between the two.

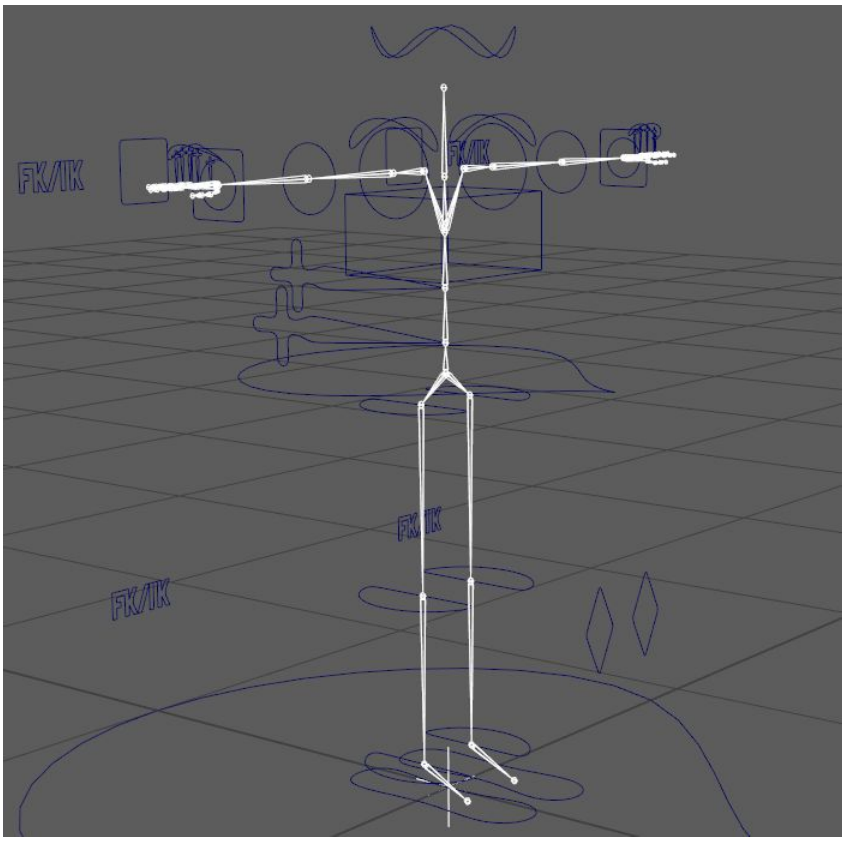
It took me a few tries to get it right due to unforeseen circumstances. One of them being the joint orientation, which apparently plays a big role in setting this system up. From my personal experience, the joints need to be orientated towards the world axis. Otherwise, when constraining the bones to each other, an offset will occur that will push the bones in directions. Usually, when creating a character, you do not model it in an a T-pose, but in an A-pose for a more organic look and deformation. At those moments, you do not want the orientation to be world orientated, but orientated down towards the next bone.

After some trials and errors, I figured out that the set up needs to be done in a very specific order and the joint orientation must absolutely not be changed after duplicating. Also mirroring joints to the other side seems to cause issues, thus it is required to repeat the whole process for the other arm.

The image below is a gif of the FK/IK switch in motion. You can find the URL to the corresponding blog post in the Annex under the headline "*FK/IK*".



On the right, you can see the controller setup that was required to create the fully functional FK/IK switch rig setup.



13.2.2. Skeletal spine

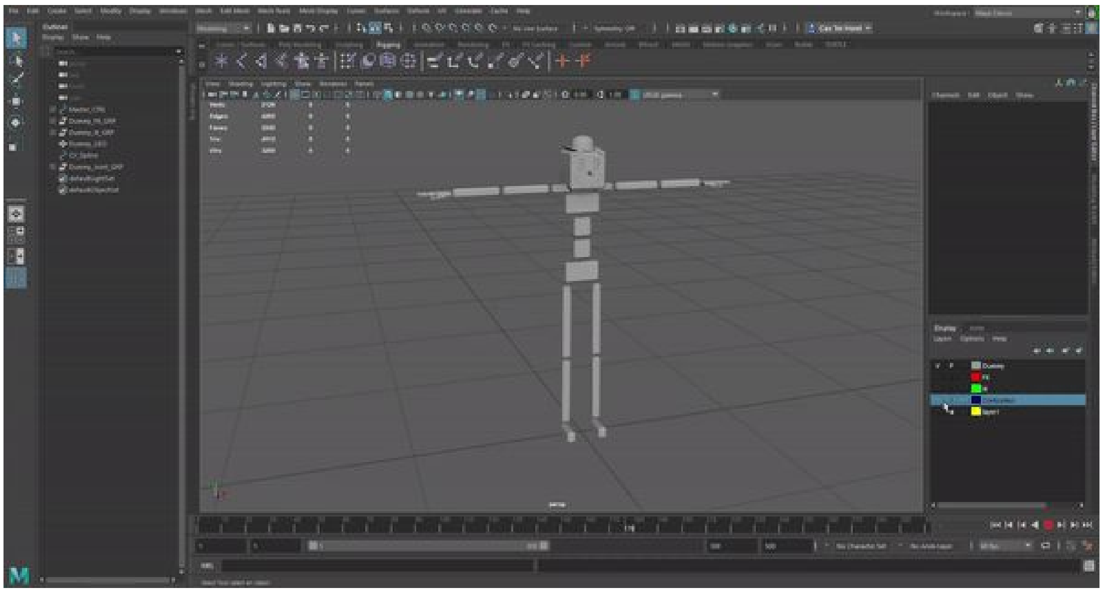
When the FK/IK switch finally fully functioned, I wanted to rig the rest of the Dummy to prepare myself for when I needed to rig the other characters, lest I make mistakes then. Better to run into the issues beforehand, while I still have time to figure out the solution. Fortunately, I did not run into any unforeseen problems, besides the one of my own making.

13.2.2.1. Squash and stretch

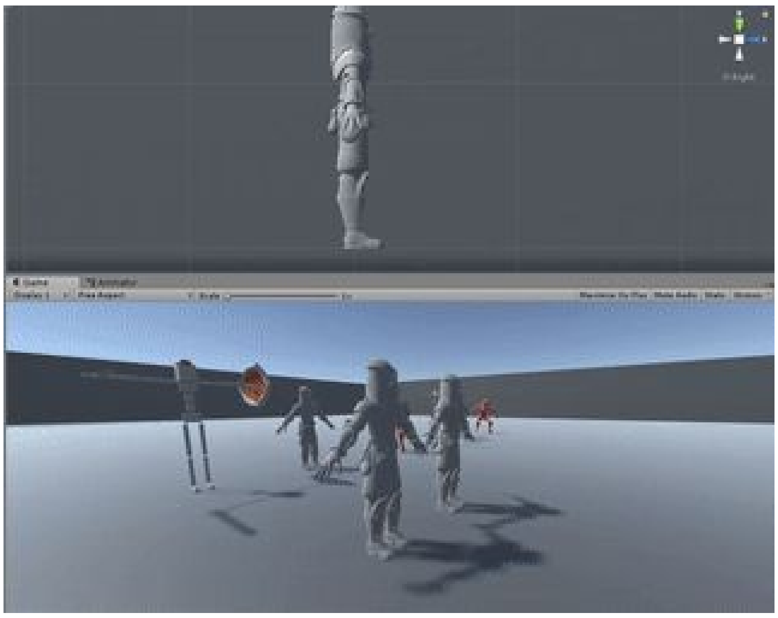
I wanted to try and make the optimal rig that could be transferred to any character and keep the same functionalities. When doing research on the spine, I came across the tutorial by an Autodesk official and I figured I could, and should, implement this into my rig. I added a squash and stretch spine system to the rig that would allow for minor deformations when moving. Normally, when things move fast, they stretch out. When things come to a sudden stop or collide with something, those things tend to squash in. The squash and stretch spine used inverse kinematics to pull or push on the effector and scale the joints on the X and Z axis depending on how far the Y axis got translated.

Unfortunately, when exporting the dummy character to Unity, I found out that Unity does not allow for joint scaling, which essentially means the squash and stretch spine is not supported. Thus I had to revert back to the original spine set up without the squash and stretch or use the forward kinematic system. But since I had changed too much during the experimentations, the rig got defective and would sink through the floor to the hips, when playing animations. The probable cause is that the root of the character got deleted and the first bone of the hierarchy took priority instead of the zero world position. A fresh rig solved the problem.

The image below is a gif of the squash and stretch spine at work. If you are not reading this digitally, you can find the URL to the corresponding blog post in the Annex, under the same headline.



The image below is a gif of the fully functioning rig at work. In the background you can see examples of the previous iterations. If you are not reading this digitally, you can find the URL to the corresponding blog post in the Annex, under the same headline.



13.2.3. Tutorials

Now that the rig is fully functional within Unity. The first thing that I wanted to research, is how the inverse kinematics system works in Unity. I imagined the best way for me to learn to understand the system is to follow multiple tutorials, but just watching the tutorials isn't the way to learn something. Thus, I created an Unity scene and did exactly as was instructed in the video.

The first tutorial I followed was from a YouTuber named Sharp Accent. An Eastern-European man with, as you might guess, a sharp English accent. Thanks to him, I learned the basics of inverse kinematics. But not only how it works, when it works right. But also how it works, when it ìsn't done right.
(Accent, S. (2015, May 28). Unity 5 Tutorial The Built-In IK System)

One of the first things I realized, was that the system can work in multiple different methods and is much more complex than my first guess. The tutorial explained 3 different uses of inverse kinematics to me, namely: Foot IK, Look IK and Arm IK. Each method uses the same built in functions, but achieves a different result, depending on what you want to use it for.

13.2.3.1. Scripting

I began with the Foot IK, since the majority of the tutorials I found were about that topic. I figured, I should familiarize myself with the topic and the code first, before I try to tackle a more difficult subject. I wanted to use the IK system for the feet, so that the character could step up on ledges. Our Unity terrain in the game consist of hexagonal tiles with each a different height. Instead of a smooth, even terrain, thus beginning with the feet seemed like a good place to start, since it a fairly important aspect of the game.

The custom code I wrote myself, following the tutorials, can be found in the Annex under the headline "*Foot IK*". The result of my code came in the ballpark of what I wanted it to do. There was no snapping of the limbs and the movement looked somewhat natural, but only when walking on slopes and uneven terrain. Unfortunately, it didn't quite work right on stairs and ledges. The feet would move through the ledge, before smoothly placing it's foot on top of the ledge when the value changed from 0 to 1.

This was due to the fact that the code did not use any prediction. It only placed the feet on the position where the downward raycast hit. It didn't know when a ledge would come until it was already too late. What I really wanted to happen was for the foot to raise itself and step up upon the ledge, in a somewhat realistic manner.

I added a slight offset to the Y axis of the feet, because the point from where the ray is being cast, is the foot joint. Which basically means that it's being cast from the ankle and not the bottom of the foot.

Unfortunately, trying to figure out when a ledge is coming, is far more difficult than I anticipated. In simple terms, by shooting a raycast forward, as well as downward, you could figure out when a ledge is coming, but combining both and place the foot accordingly, seemed nearly impossible for a novice programmer like myself. Even the programmers in my team had a hard time understanding the code and figuring out a solution. There are multiple examples of people getting this Foot IK system right, with prediction implemented, but no tutorials explaining it. Only assets that can be bought in the Asset Store or videos showcase the script.

The downside of my code, is that the raycast is always on. Meaning that the feet will always be glued to the floor. Therefore, I had to add an event curve and a float parameters to the animation itself. Whenever the foot would lift itself off of the ground, I set the value to 0. Whenever it would touch the floor, it’s automatically set to 1.

13.2.3.2. Experimentations

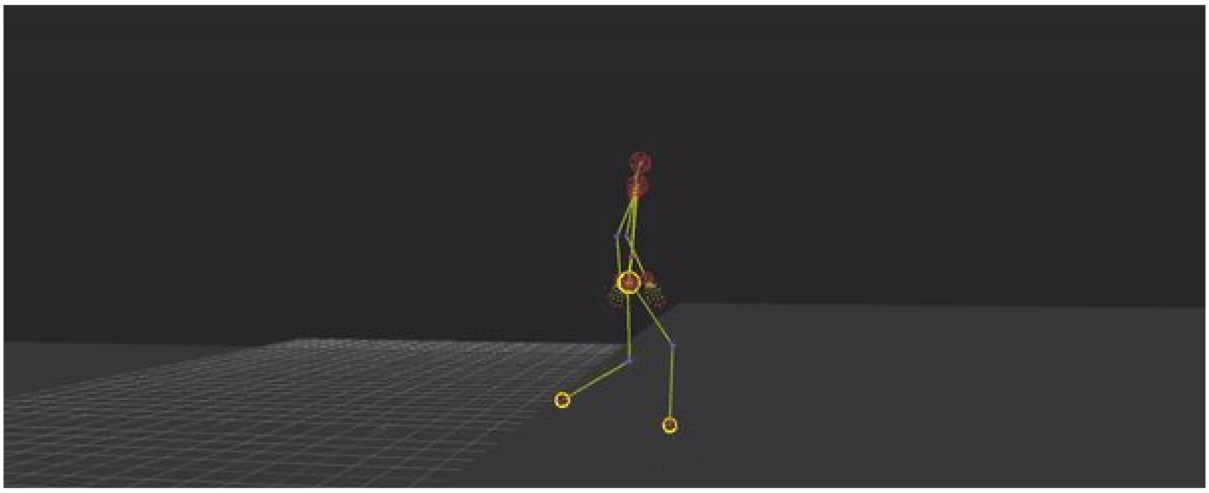
Trying to think of an original solution for the Foot IK problem was hard. At some point I had given up on the Foot IK, because after spending weeks of trying and messing around with the code, it simply wouldn’t work. I decided to let it be for some time and look for some free code online. I still wanted the Foot IK to be in the game, but I did not want to waste anymore time on it, since it seemed futile.

I found 2 promising scripts online. It wasn’t explained how the code worked, but only how to set it up in Unity. Both scripts had the same issue, but were they far more elaborate than my own code and gave much better results. Although the legs snapped into place, instead of transitioning smoothly. Looking at the code, I tried to understand how it functioned. I got the gist of it, but it is very complex, with a lot of functions I hadn’t used in my own code.

One of the authors of the scripts was kind enough to leave a detailed description. When comparing the reference code with my own, and trying to integrate some parts of it, proved a difficult task. I asked one the programmers if he could help me reverse engineer the code, but make the transitions of the feet happen smoothly. But after days of trying, it was still a futile attempt.

Both Scripts can be found in the Annex under the headline “*Reference Foot IK 1*” and “*Reference Foot IK 2*”.

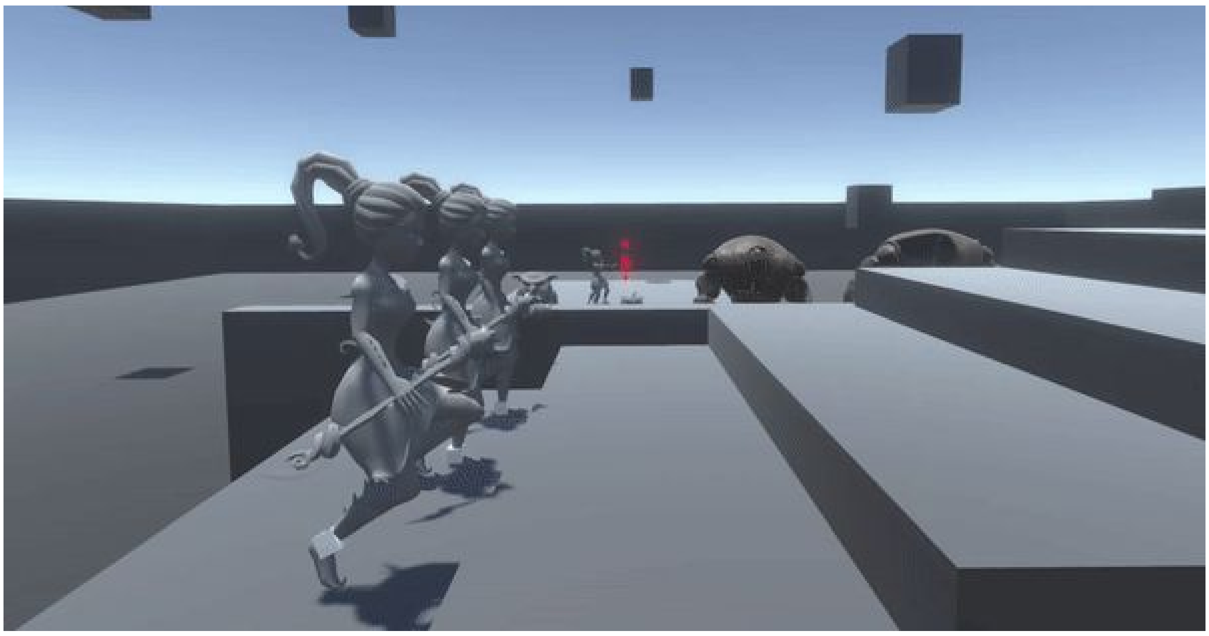
When discussing the situation with one of the programmers, an idea sparked in my mind, which seemed like a logical workaround for the foot IK. The idea was, that instead of trying to predict when the a ledge is coming, and move the foot upwards accordingly, I needed to make 2 separate animations, that would play when the characters was about to step up the ledge. The programmers already had a system in place that calculated the position of the character in the world and knew the height of the next tile.



I made an animation for each foot, depending on which leg was in front, so the right step-up animations could be played. Then we'd simply would pull down the foot by the amount it had overshot and translate the root of the character by that amount as well, to create a smooth transition. Unfortunately, it once again proved very difficult to write the code, but it worked to a certain extent.

One of the side effects of using the step up animation, was that the character launched into the air for a moment and dragged it's foot behind it. This happened because the root of the character overshot and was pulled down too slowly.

The image below is a gif of my custom script and the step up animation at work. The URL for the corresponding can be found in the Annex under the headline: "*Step up*". The first character is driven by my scripts, the 2 behind are being driven by other scripts that show even worse results. The script needs more tweaking and a lengthy analysis of what is causing the issues, but it could possibly work. It seems to be an ineffective workaround, though.



13.2.4. Arm IK

When starting to work on the inverse kinematics for the arms, I first needed to figure out how I should set up the rig properly so that the character could hold a weapon. I imagined that I could insert a 2-bone joint chain in the weapon. One bone that functions as the root, to be constrained to the hand. The other bone would be point where the weapon would collide with the target. I didn't fully understand how inverse kinematics worked in Unity yet, but I understood the basic concept.

From what I understood, the last bone in the chain was the effector. By making the weapon a part of the rig, as a continuous chain or a separate bone with a parent-child constraint relation, I figured I could assign the effector myself. Unfortunately, Unity's built-in function does not allow for that. The system only works on humanoid characters and can only call upon the preset effectors, or AvatarIKGoal's as Unity's documentation describes it: (Technologies, U. (n.d.). AvatarIKGoal). Such as the hands and feet, as well as their adjourned joints, which are the elbow and knee.

Since only the preset effectors could be used, I figured I should get rid of the bone in the weapon, since it wasn't doing much good anyway. I parented the weapon to the hand joint instead.

13.2.4.1. Sword slashing

After having some experience from working with the Foot IK, I figured the arm IK would be an easier process and it turned out the be easier, except for the animation I chose to use. Which is a sideward slashing animation as you can see on the right.

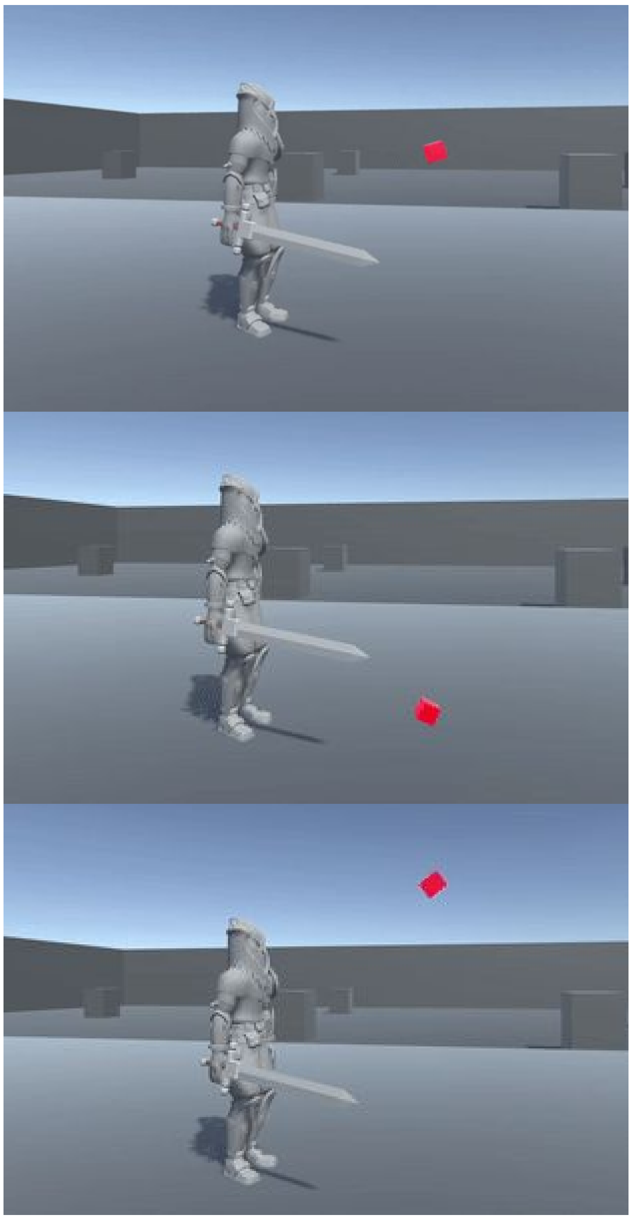
The code works quite well for any kind of attack animation, besides animations which have a side sweeping motion, due to the lingering influence of the target. The influence of the target is decided by a float parameter with a curve.
Ideally, the influence gradually increases overtime until the sword reaches the target, then immediately disappears and the override blends into the original animation.

It took me some time trying to figure out how to smoothly transition between the override and the original animation. Adjusting the curves seemed the most obvious thing to do, thus I tried various options to find the perfect blending point, but nothing seemed to erase that last lingering moment. One of the programmers suggested to cut the animation into two. One animation that charges up and slices the target and another that continues where the other left off. We figured it might be easier to blend between two animations rather than trying to smoothly decrease the influence. But alas, the blending of animations seemed too difficult for us to write in code, so we had to go back to square one.

I took some time off from trying to figure out how to smoothly blend the animations and focused on the animation itself, because it still looked awful. The whole point of using the IK system was to create animations that look natural whatever the target is, but when the target goes too high or low, it looks wrong how the spine doesn’t bend as well.

I found a function, similar to inverse kinematics, called LookAt. Which allows for the head and spine to deform depending on the position of a target. By implementing this function in the script and assigning the same target, the animation becomes a lot more natural looking and somehow it fixed some of the lingering of the weapon.

You can find the corresponding code with green highlighted explanations in the Annex, under the headline "*Slashing IK*". As well as the URL to the corresponding blog post in the Annex, under the same headline.



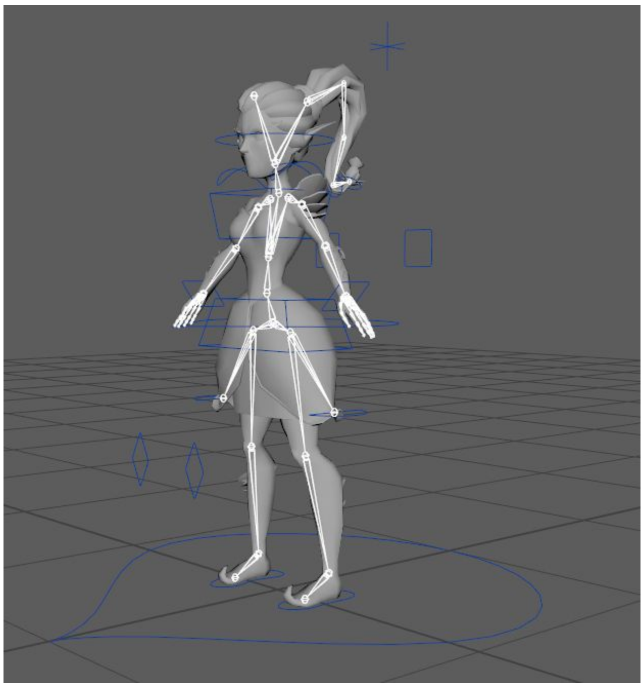
13.2.5. Other work

Besides working on inverse kinematics and scripting, I was required to do some other work to help the project team succeed.

13.2.5.1. Rigging

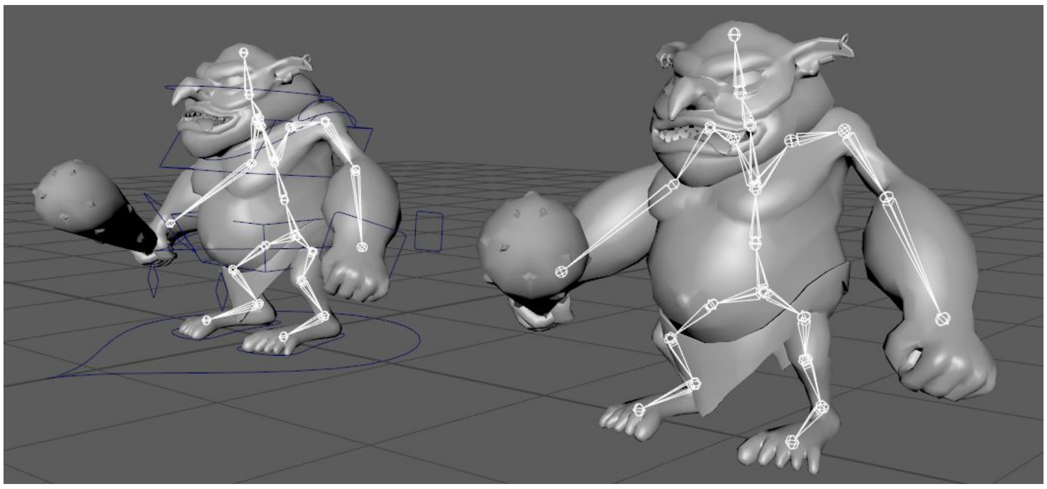
I’ve rigged all the characters currently in game, which are:

- The plant girl (role: healer)
- The owl (role: scout)
- The golem (role: tank)
- The thief (role: assassin)
- The goblin (role: enemy)



The plant girl, owl and goblin are fully rigged within Maya with the FK/IK controller setup. The other characters just have a basic joint setup.

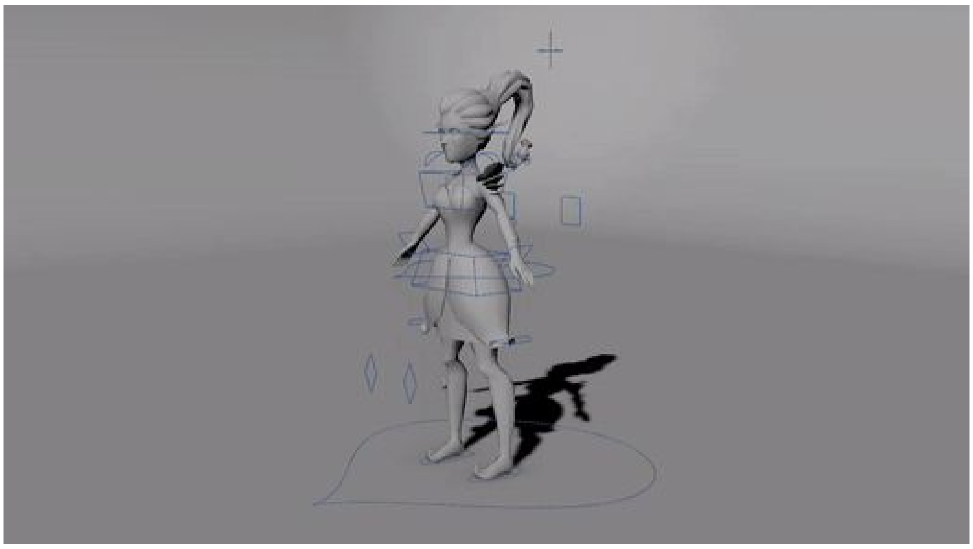
Below is an image of the goblin with its fully functional rig.



13.2.5.2. Weightpainting

I’ve weight painted the plant girl as well as the goblin.

The image below is a gif that demonstrates the polished weightpaint. You can find the URL to the corresponding blog post in the Annex, under the “*Weightpaint*” headline.

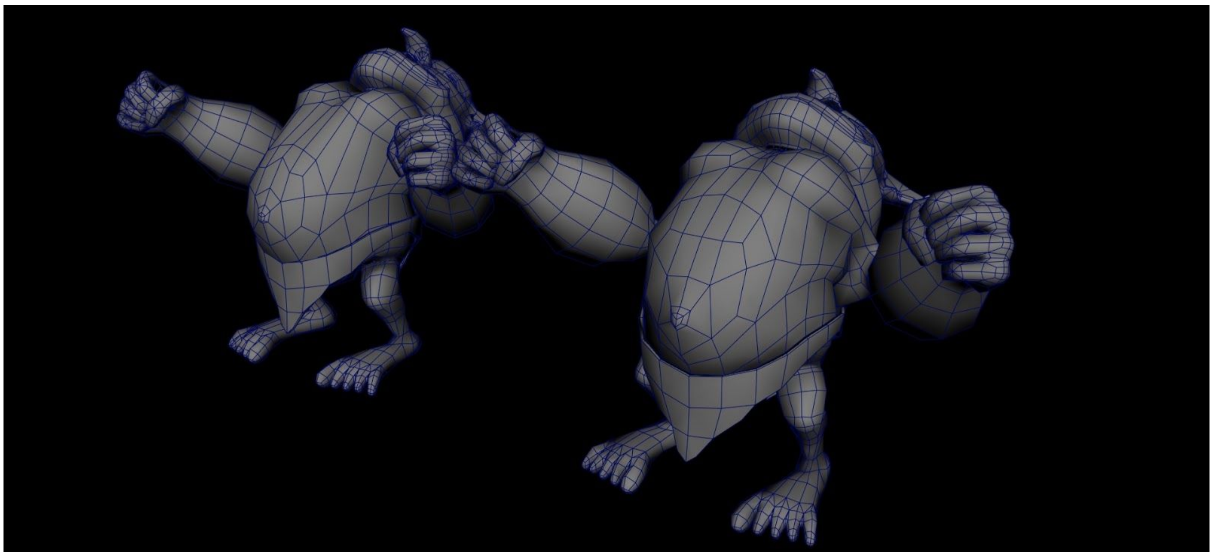


13.2.5.3. Delta Mush

I’ve done research on what the fastest way is to skin and weightpaint a model. I knew from previous experiences that the Delta Mush deformer tool in Maya acts as a second skin bind, which helps deform the model better when animation, but the deformer tool cannot be exported to another program like Unity and therefore needs to be baked onto a copy of itself. It needs some minor polish afterwards, because the baked copy isn’t an exact rendition of the original.

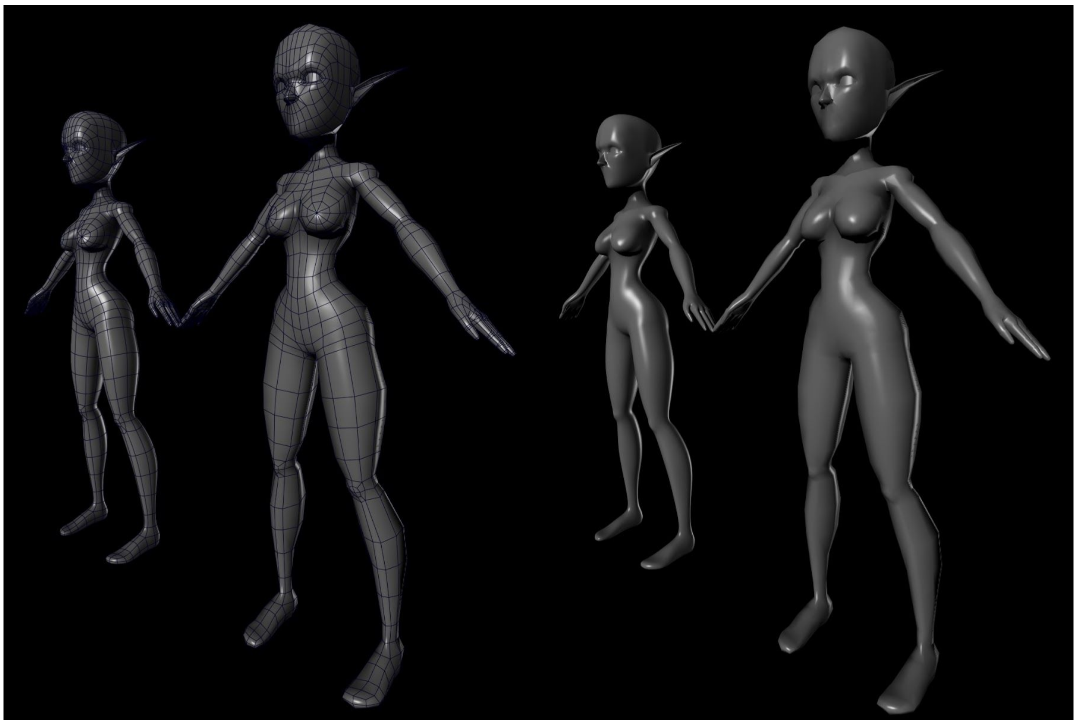
Using delta mush saves a lot of time on bigger chunks of the model, but the finer detail still needs to be hand painted.

Below is an image of the goblin with (right) and without (left) the delta mush enabled.



13.2.5.4. Modelling

I’ve reconstructed the plant girl in to a female base mesh for the other team members to use as a template for their characters. We have multiple female characters in our game and making sure they all have the same proportions will improve the overall aesthetic.



13.2.5.5. Animations

13.2.5.5.1. Plant girl

I've made several animations for the plant girl and owl. These animations are taken from the website: Mixamo and adjusted to fit our characters. The animations for the scout received more polish than the plant girl, since there weren't any animations on the site that matched the abilities, thus I mixed multiple samples and adjusted them.

Below you can see the animations for the plant girl. There are 6 animations in total, for 4 abilities.

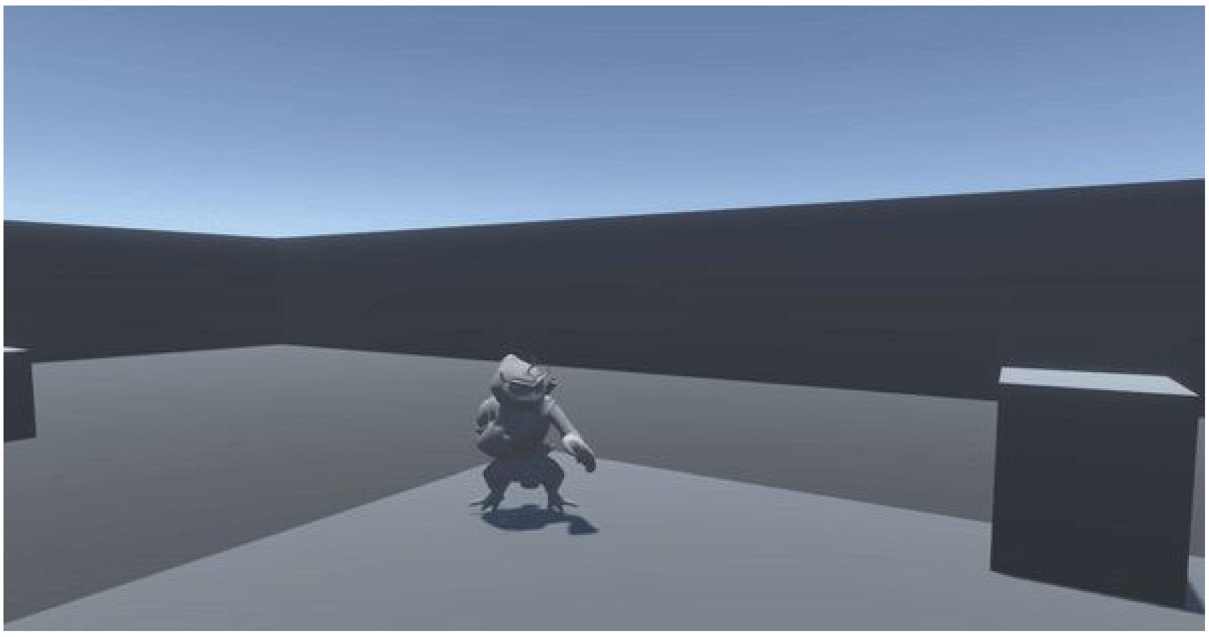
1. Basic attack
2. Basic heal
3. Summoning roots (trap)
4. Burrowing underground
5. Burrowing back up
6. Revive

You can find the URL to the corresponding blog post in the Annex, under the "*Plant girl*" headline.

13.2.5.5.2. Scout

Below you can see the animations for the scout. There are 6 animations in total, for 3 movement and 3 abilities. Some in-between animations still need to be made to properly transition between the animations.

1. Running
2. Flying forward
3. Casting spell while flying
4. Flying backwards
5. Whirlwind slash
6. Windwall



You can find the URL to the corresponding blog post in the Annex, under the "*Scout*" headline.

13.2.6. Team role

This project team consists of a lot of people from a lot of different studies, we started with 16 and will end with 15, since one of us has already graduated halfway throughout the project. Everyone has a background in arts & technology or some kind of prior knowledge of entertainment or computer engineering. Everyone has a creative background to say the least. But only a few, myself included, have a game design background and know the entire process from front to back. Which resulted in me and my fellow game designers to take up the role of project leaders.

I declined the honor, because I was not interested in doing all the tasks that come with being a project leader, but I have functioned as a quality assurer and overall teacher to the ones who were new to the process. I've taught people how to 3D model, UV-map, weightpaint and optimize the general workflow by showing them my methods.

Once a models was done, according to the creators, I usually took a look at them to see if they were animation worthy. If not, I would send them back, or tweak the model slightly myself if my request was too difficult for them. I also took a look at the UV-map to see if it could be optimized a bit more.

When the models were finalized, I rigged them with a basic joint set up and send them back to their creators to weight paint them. I would usually keyframe a few extreme poses to help them along.

14. Annex

14.1. Blog links

Changeable Walk

Ter Horst, C. (2018, November 28). Sprint 6 | Animations | Weapon override & retargeting. Retrieved from https://tabletoprpgblog.wordpress.com/2018/11/28/sprint-6-animations-weapon-override-retargeting

FK/IK

Ter Horst, C. (2018, October 11). Sprint 1 | Research | IK/FK switch. Retrieved from https://tabletoprpgblog.wordpress.com/2018/09/24/sprint-1-research-ik-fk-switch

Squash & Stretch spine

Ter Horst, C. (2018, October 11). Sprint 1 | Research | Dummy Rig. Retrieved from https://tabletoprpgblog.wordpress.com/2018/09/28/sprint-1-research-dummy-rig

Fully functioning rig

Ter Horst, C. (2018, October 19). Sprint 3 | Research | Rig fixed. Retrieved from https://tabletoprpgblog.wordpress.com/2018/10/12/sprint-2-research-rig-fixed

Step up

Ter Horst, C. (2018, November 28). Sprint 6 | Research | Foot IK. Retrieved from https://tabletoprpgblog.wordpress.com/2018/11/28/sprint-6-research-foot-ik

Slashing

Ter Horst, C. (2018, November 07). Sprint 5 | Research | Sword Slash. Retrieved from https://tabletoprpgblog.wordpress.com/2018/11/07/sprint-5-research-sword-slash

Weight paint

Ter Horst, C. (2018, November 22). Sprint 5 | 3D modeling | Weight painting. Retrieved from https://tabletoprpgblog.wordpress.com/2018/11/22/sprint-5-3d-modeling-weight-painting

Slashing IK

Ter Horst, C. (2018, November 16). Sprint 5 | Research |Procedural animation (Slashing). Retrieved from https://tabletoprpgblog.wordpress.com/2018/11/16/sprint-5-research-procedural-animation-slashing

Plant girl

Ter Horst, C. (2018, December 04). Sprint 6 | Animations | Plant Girl Abilities. Retrieved from https://tabletoprpgblog.wordpress.com/2018/12/04/sprint-6-animations-plant-girl-abilities

Scout

Ter Horst, C. (2018, December 18). Sprint 7 | Animations | Scout. Retrieved from https://tabletoprpgblog.wordpress.com/2018/12/18/sprint-7-animations-scout

14.2. Scripts

14.2.1. Foot IK

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

public class FootIK1 : MonoBehaviour
{

    Animator anim;

    public float ikWeight = 1;
    public LayerMask rayMask;

    /*
    public Transform leftIKTarget;
    public Transform rightIKTarget;

    public Transform hintLeft;
    public Transform hintRight;
    */

    Vector3 lFpos;
    Vector3 rFpos;

    Quaternion lFrot;
    Quaternion rFrot;

    float lFWeight;
    float rFWeight;

    Transform leftFoot;
    Transform rightFoot;
    Transform hip;

    public float FootOffsetY;

    // Use this for initialization
    void Start()
    {
        anim = GetComponent<Animator>();

        leftFoot = anim.GetBoneTransform(HumanBodyBones.LeftFoot);
        rightFoot = anim.GetBoneTransform(HumanBodyBones.RightFoot);

        lFrot = leftFoot.rotation;
        rFrot = rightFoot.rotation;

    }

    // Update is called once per frame
    void Update()
    {
        RaycastHit leftHit;
        RaycastHit rightHit;

        Vector3 lpos = leftFoot.TransformPoint(Vector3.zero);
        Vector3 rpos = rightFoot.TransformPoint(Vector3.zero);

        Debug.DrawRay(lpos, Vector3.down, Color.green);
        if (Physics.Raycast(lpos, Vector3.down, out leftHit, rayMask.value))
        {
            lFpos = leftHit.point;
            lFrot = Quaternion.FromToRotation(transform.up, leftHit.normal) *
transform.rotation;
        }

        Debug.DrawRay(rpos, Vector3.down, Color.red);
        if (Physics.Raycast(rpos, -Vector3.up, out rightHit, rayMask.value))
        {
            rFpos = rightHit.point;
            rFrot = Quaternion.FromToRotation(transform.up, rightHit.normal) *
transform.rotation;
        }
    }

    void OnAnimatorIK()
    {
        lFWeight = anim.GetFloat("LeftFoot");
        rFWeight = anim.GetFloat("RightFoot");

        //Position Feet
        anim.SetIKPositionWeight(AvatarIKGoal.LeftFoot, lFWeight);
        anim.SetIKPositionWeight(AvatarIKGoal.RightFoot, rFWeight);

        anim.SetIKPosition(AvatarIKGoal.LeftFoot, lFpos + new Vector3(0, FootOffsetY,
0));
        anim.SetIKPosition(AvatarIKGoal.RightFoot, rFpos + new Vector3(0, FootOffsetY,
0));

        //Rotation Feet
        anim.SetIKRotationWeight(AvatarIKGoal.LeftFoot, lFWeight);
        anim.SetIKRotationWeight(AvatarIKGoal.RightFoot, rFWeight);

        anim.SetIKRotation(AvatarIKGoal.LeftFoot, lFrot);
        anim.SetIKRotation(AvatarIKGoal.RightFoot, rFrot);

        //Position Knees
        //anim.SetIKHintPositionWeight(AvatarIKHint.LeftKnee, ikWeight);
        //anim.SetIKHintPositionWeight(AvatarIKHint.RightKnee, ikWeight);

        //anim.SetIKHintPosition(AvatarIKHint.LeftKnee, hintLeft.position);
        //anim.SetIKHintPosition(AvatarIKHint.RightKnee, hintRight.position);


    }
}
```

14.2.2. Foot IK Pro

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

public class FilmStormIK : MonoBehaviour
{
    public Animator anim; //the animator attached to the character
    private Vector3 rightFootPosition, leftFootPosition, rightFootIkPosition,
leftFootIkPosition; // used to get the positions of the feet
    private Quaternion leftFootIkRotation, rightFootIkRotation; // used to get the
rotations of the feet
    private float lastPelvisPositionY, lastRightFootPositionY, lastLeftFootPositionY;
//used to get the last known postion of the feet and hip

    [Header("Feet Grounder")] // creates a new header in the inspector
    public bool enableFeetIk = true; // turn on/off IK
    [Range(0, 2)] public float heightFromGroundRaycast = 1.5f; //height from wich ray is
cast
    [Range(0, 2)] public float raycastDownDistance = 1.5f; //length of ray
    public LayerMask environmentLayer; //what objects the ray is able to hit
    [Range(0, 1)] public float pelvisOffset = 0f; //offset of the hip
    [Range(0, 1)] public float PelvisUpAndDownSpeed = 0.5f; //how fast the hips translate
to IK offset
    [Range(0, 1)] public float feetToIkPositionSpeed = 0f; //how fast the feet translate
to IK offset

    public string leftFoot = "LeftFoot"; // gets the animation curves
    public string rightFoot = "RightFoot";

    public bool useRotation = false; // enables foot IK rotation
    public bool showSolverDebug = true; //show raycast line in viewport

    // Use this for initialization
    void Start ()
    {
        anim = GetComponent<Animator>();
    }

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

       }

    private void FixedUpdate()
    {
        if(enableFeetIk == false) { return; } // if IK is not active, do nothing
        if(anim == null) { return; } // if there is no animator, do nothing

        AdjustFeetTarget(ref rightFootPosition, HumanBodyBones.RightFoot);  //do this
function, updates the position of the feet | gets the correct foot from the skeleton
        AdjustFeetTarget(ref leftFootPosition, HumanBodyBones.LeftFoot); //ref allows you
to reference a value of another function

        FeetPositionSolver(rightFootPosition, ref rightFootIkPosition, ref
rightFootIkRotation); //do this function, finds the current position and rotation of the
feet and raycast
```

```
        FeetPositionSolver(leftFootPosition, ref leftFootIkPosition, ref
leftFootIkRotation);

    }

    private void OnAnimatorIK()
    {
        if (enableFeetIk == false) { return; }
        if (anim == null) { return; }

        MovePelvisHeight(); //do this function. always running

        //right foot IK pos and rot
        anim.SetIKPositionWeight(AvatarIKGoal.RightFoot, 1); // set the inflience of the
IK weight. Always on

        if(useRotation)
        {
            anim.SetIKRotationWeight(AvatarIKGoal.RightFoot, anim.GetFloat(rightFoot));
//set the influence of the IK weight depending on the animation curve.
        }

        MoveFeetToIkPoint(AvatarIKGoal.RightFoot, rightFootIkPosition,
rightFootIkRotation, ref lastRightFootPositionY); //Moves the right foot down to the
ground and rotate it according the the normal

        //left foot IK pos and rot -- enable pro features
        anim.SetIKPositionWeight(AvatarIKGoal.LeftFoot, 1);

        if (useRotation)
        {
            anim.SetIKRotationWeight(AvatarIKGoal.LeftFoot, anim.GetFloat(leftFoot));
        }

        MoveFeetToIkPoint(AvatarIKGoal.LeftFoot, leftFootIkPosition, leftFootIkRotation,
ref lastLeftFootPositionY);
    }

    void MoveFeetToIkPoint ( AvatarIKGoal foot, Vector3 positionIkHolder, Quaternion
rotationIkHolder, ref float lastFootPositionY)
    {
        Vector3 targetIkPosition = anim.GetIKPosition(foot);

        if (positionIkHolder!= Vector3.zero) //if it's not 0,0,0
        {
            targetIkPosition = transform.InverseTransformPoint(targetIkPosition);
//transfroms the position of the target from world space to local space
            positionIkHolder = transform.InverseTransformPoint(positionIkHolder);
//transfroms the position of the foot to local space

            float yVar = Mathf.Lerp(lastFootPositionY, positionIkHolder.y,
feetToIkPositionSpeed); // take the last position of the foot and move it over time
            targetIkPosition.y += yVar; //

            lastFootPositionY = yVar;
            targetIkPosition = transform.TransformPoint(targetIkPosition);

            anim.SetIKRotation(foot, rotationIkHolder);
        }

        anim.SetIKPosition(foot, targetIkPosition);
    }

    private void MovePelvisHeight()
    {
        if(rightFootIkPosition == Vector3.zero || leftFootIkPosition == Vector3.zero ||
lastPelvisPositionY == 0)
        {
            lastPelvisPositionY = anim.bodyPosition.y;
            return;
        }

        float lOffsetPosition = leftFootIkPosition.y - transform.position.y;
        float rOffsetPosition = rightFootIkPosition.y - transform.position.y;

        float totalOffset = (lOffsetPosition < rOffsetPosition) ? lOffsetPosition :
rOffsetPosition;

        Vector3 newPelvisPosition = anim.bodyPosition + Vector3.up * totalOffset;
        newPelvisPosition.y = Mathf.Lerp(lastPelvisPositionY, newPelvisPosition.y,
PelvisUpAndDownSpeed);

        anim.bodyPosition = newPelvisPosition;

        lastPelvisPositionY = anim.bodyPosition.y;
    }

    /// <summary>
    /// Raycasting
    /// </summary>
    private void FeetPositionSolver (Vector3 fromSkyPosition, ref Vector3 feetIkPosition,
ref Quaternion feetIkRotation)
    {
        RaycastHit feetOutHit;
        if (showSolverDebug)
            Debug.DrawLine(fromSkyPosition, fromSkyPosition + Vector3.down *
(raycastDownDistance + heightFromGroundRaycast), Color.red);

        if (Physics.Raycast(fromSkyPosition, Vector3.down, out feetOutHit,
raycastDownDistance + heightFromGroundRaycast, environmentLayer))
        {
            feetIkPosition = fromSkyPosition;
            feetIkPosition.y = feetOutHit.point.y + pelvisOffset;
            feetIkRotation = Quaternion.FromToRotation(Vector3.up, feetOutHit.normal) *
transform.rotation;

            return;
        }

        feetIkPosition = Vector3.zero;
    }

    private void AdjustFeetTarget (ref Vector3 feetPositions, HumanBodyBones foot)
    {
        feetPositions = anim.GetBoneTransform(foot).position;
        feetPositions.y = transform.position.y + heightFromGroundRaycast;


    }
}
```

14.2.3. Reference Foot IK 1

```
using UnityEngine;
using System.Collections;

public class FootIKProgramYourFace : MonoBehaviour
{
    public LayerMask rayMask;
    public float baseOffset;
    public float alignSpeed = 5.0f;
    public bool automaticFixY = true;

    float lIKh;
       float rIKh;
    float lIKw;
    float rIKw;
       bool lHit;
       bool rHit;
       bool useLIK;
       bool useRIK;
       Vector3 lNrm;
       Vector3 rNrm;

    bool groundHit;
    float groundHeight;
    RaycastHit groundInfo;

    bool grounded;
       Animator anim;
       CharacterController controller;

       void Start()
       {
              anim = GetComponent<Animator>();
              controller = GetComponent<CharacterController>();
       }

       void OnAnimatorIK(int layer)
       {
        GatherGroundInfo();

        if (!groundHit)
            return;

        RayCastLeg(AvatarIKGoal.LeftFoot);
        RayCastLeg(AvatarIKGoal.RightFoot);

        Transform leftHeel = anim.GetBoneTransform(HumanBodyBones.LeftFoot);
              Transform rightHeel = anim.GetBoneTransform(HumanBodyBones.RightFoot);



        if(!lHit) lIKh = leftHeel.position.y;
              if(!rHit) rIKh = rightHeel.position.y;

        SetFootPosition(lHit, AvatarIKGoal.LeftFoot, leftHeel.position, lNrm, lIKh, ref
lIKw);
        SetFootPosition(rHit, AvatarIKGoal.RightFoot, rightHeel.position, rNrm, rIKh, ref
rIKw);
```

```
        Debug.Log("LEFT WEIGHT: " + lIKw + "    RIGHT WEIGHT: " + rIKw);
       }

    void LateUpdate()
    {
        if(automaticFixY)
            FixY();
    }

    private void GatherGroundInfo()
    {
        Vector3 cBase = transform.TransformPoint(controller.center) + Vector3.down *
(controller.height * 0.5f - controller.radius);
        groundHit = Physics.SphereCast(cBase, controller.radius, Vector3.down, out
groundInfo, Mathf.Infinity, rayMask.value);
        groundHeight = cBase.y - groundInfo.distance - controller.radius;
    }

       private void RayCastLeg(AvatarIKGoal ag)
       {
              if(ag == AvatarIKGoal.LeftHand || ag == AvatarIKGoal.RightHand) return;
              bool h = false;
        float baseHeight = GetBaseHeight();
        float rayheight = (transform.TransformPoint(controller.center).y -
(controller.height * 0.5f - controller.radius)) - baseHeight;

        RaycastHit hit;
              Transform heel = anim.GetBoneTransform(ag ==
AvatarIKGoal.LeftFoot?HumanBodyBones.LeftFoot:HumanBodyBones.RightFoot);
              Vector3 heelPos = heel.position;
        heelPos.y = groundHeight + rayheight;

        //Forward offset, so we can see steps ahead of time
        //heelPos += transform.forward * 0.2f;
        //rayheight *= 0.2f;

        heelPos.y = groundHeight + rayheight;

        Debug.DrawRay(heelPos, Vector3.down * rayheight, Color.red);

              if(Physics.Raycast(heelPos,Vector3.down,out hit, rayheight,rayMask.value))
              {
                     h = true;
                     if(ag == AvatarIKGoal.LeftFoot)
                     {
                            lHit = true;
                            lIKh = hit.point.y;
                            lNrm = hit.normal;
                     }
                     else
                     {
                            rHit = true;
                rIKh = hit.point.y;
                            rNrm = hit.normal;
                     }
              }
              if(!h)
              {
                    if(ag == AvatarIKGoal.LeftFoot) lHit = false;
                    else rHit = false;
             }
      }

    private void SetFootPosition(bool use, AvatarIKGoal ag, Vector3 heelPos, Vector3 nrm,
float IKh, ref float weight)
      {
             if(ag == AvatarIKGoal.LeftHand || ag == AvatarIKGoal.RightHand) return;
        weight += (use ? 1.0f : -1.0f) * Time.deltaTime * alignSpeed;
        weight = Mathf.Clamp01(weight);

             if(use)
             {
            Vector3 rotAxis = Vector3.Cross(Vector3.up, nrm);
            float angle = Vector3.Angle(Vector3.up, nrm);
            Quaternion rot = Quaternion.AngleAxis(angle * weight, rotAxis);
            anim.SetIKRotationWeight(ag, weight);
            anim.SetIKRotation(ag, rot * anim.GetIKRotation(ag));

            float baseHeight = GetBaseHeight();
            float animHeight = (heelPos.y - baseHeight) / (rot * Vector3.up).y;
            Vector3 pos = new Vector3(heelPos.x, Mathf.Max(IKh, baseHeight) + animHeight,
heelPos.z);
                    anim.SetIKPositionWeight(ag, weight);
                    anim.SetIKPosition(ag, pos);
             }
             else
             {
            anim.SetIKPositionWeight(ag, weight);
            anim.SetIKRotationWeight(ag, weight);
        }
      }

    public void FixY()
    {
        if (!groundHit)
            return;

        float min = groundHeight;
        if (lHit && lIKh < min)
            min = lIKh;

        if (rHit && rIKh < min)
            min = rIKh;

        Vector3 pos = transform.position;
        if (GetBaseHeight() <= min)
        {
            pos.y = min - baseOffset;
            transform.position = pos;
            grounded = true;
        }
        else
            grounded = false;
    }

      public bool isGrounded()
```

```
        {
                Vector3 tempNRM;
                return isGrounded(out tempNRM);
        }

        public bool isGrounded(out Vector3 nrm)
        {
                nrm = Vector3.up;
         if (!groundHit)
             return false;

         nrm = groundInfo.normal;
         return grounded;
    }

    private float GetBaseHeight()
    {
         return transform.position.y + baseOffset;
    }
}
```

14.2.4. Reference Foot IK 2

```
using UnityEngine;
using System;

[RequireComponent(typeof(Animator))]

//Name of class must be name of file as well

public class FootIKDownload : MonoBehaviour {

        protected Animator avatar;
        public bool ikActive = false;
        //To control how much will affect the transform of the IK
        public float transformWeigth = 1.0f;
        //To make change the value smoothly
        public float smooth = 10;
        //The position of my left foot
        public Transform footL;
        //A Offset to the foot not jam on the floor
        public Vector3 footLoffset;
        //I will use to control when affect the position during animation
        public float weightFootL = 1;
        //The position of my right foot
        public Transform footR;
        //A Offset to the foot not jam on the floor
        public Vector3 footRoffset;
        //I will use to control when affect the position during animation
        public float weightFootR;
        //I'll save the Raycast hit position of the feet
        private Vector3 footPosL;
        private Vector3 footPosR;
        //To access my Collider
        private CapsuleCollider myCollider;
        //Default [center] of my collider
        private Vector3 defCenter;
        //Default [Height] of my collider
        private float defHeight;
        //[LayerMask] to define with layer my foot [RayCast] will collide
        public LayerMask rayLayer;

    public float rayOffset = 0.1f;

        // Use this for initialization
        void Start ()
        {
                //Set the component
                avatar = GetComponent<Animator>();
                myCollider = GetComponent<CapsuleCollider>();
                //Save the values
                defCenter = myCollider.center;
                defHeight = myCollider.height;
        }

        void OnAnimatorIK(int layerIndex)
        {
                //If the [avatar] value is set
                if(avatar)
                {
                        //If [ikActive] is [true]
```

```
                    if(ikActive)
                    {
                           //Change the [transformWeigth] value to 1 smoothly
                           if(transformWeigth != 1.0f){
                                  transformWeigth = Mathf.Lerp(transformWeigth, 1.0f, 
Time.deltaTime * smooth);
                                  //If the value [transformWeigth] be greater than 0.99 
it will be 1
                                  if(transformWeigth >= 0.99){w
                                         transformWeigth = 1.0f;
                                  }
                           }
                           //If the situation of the player is [Idle]
                           if(avatar.GetBool("OnGround") == false)
               {
                   //Set how much will affect the IK [transform]
                   //Debug.Log("ASDASd");

       avatar.SetIKPositionWeight(AvatarIKGoal.LeftFoot,transformWeigth);

       avatar.SetIKRotationWeight(AvatarIKGoal.LeftFoot,transformWeigth);

       avatar.SetIKPositionWeight(AvatarIKGoal.RightFoot,transformWeigth);

       avatar.SetIKRotationWeight(AvatarIKGoal.RightFoot,transformWeigth);

                                  IdleIK();
                           }
                           //If the situation of the player is [Walk] or [Run]
                           else if(avatar.GetBool("OnGround") == true)
               {
                   //Debug.Log("ASDASd");
                   weightFootL = avatar.GetFloat("LeftFoot");
                   weightFootR = avatar.GetFloat("RightFoot");

                   //Set how much will affect the IK [transform]
                   avatar.SetIKPositionWeight(AvatarIKGoal.LeftFoot,transformWeigth * 
weightFootL);

       avatar.SetIKRotationWeight(AvatarIKGoal.LeftFoot,transformWeigth * weightFootL);

       avatar.SetIKPositionWeight(AvatarIKGoal.RightFoot,transformWeigth * weightFootR);

       avatar.SetIKRotationWeight(AvatarIKGoal.RightFoot,transformWeigth * weightFootR);

                                  WalkRunIK();
                           }
                    }
                    //If [ikActive] is not [true]
                    else
                    {
                           //Change the [transformWeigth] value to 0 smoothly
                           if(transformWeigth != 0.0f){
                                  transformWeigth = Mathf.Lerp(transformWeigth, 0.0f, 
Time.deltaTime * smooth);
                                  //If the value [transformWeigth] be less than 0.01 it 
will be 0
                                  if(transformWeigth <= 0.01){
```

```
                                                    transformWeigth = 0.0f;
                                             }
                                    }
                                    //Set how much will affect the IK [transform]

       avatar.SetIKPositionWeight(AvatarIKGoal.LeftFoot,transformWeigth);

       avatar.SetIKRotationWeight(AvatarIKGoal.LeftFoot,transformWeigth);

       avatar.SetIKPositionWeight(AvatarIKGoal.RightFoot,transformWeigth);

       avatar.SetIKRotationWeight(AvatarIKGoal.RightFoot,transformWeigth);
                      }
               }
       }
       void IdleIK(){
              //Create this value to use the [RaycastHit]
              RaycastHit hit;
              //Get the current position of the left foot
              footPosL = avatar.GetIKPosition(AvatarIKGoal.LeftFoot);
              //[RayCast] to the ground, to know the distance
              if(Physics.Raycast(footPosL + Vector3.up, Vector3.down, out hit, 2.0f,
rayLayer))
              {
                     //Show [Ray]
                     Debug.DrawLine(hit.point, hit.point + hit.normal, Color.yellow);
                     //Set the new position of IK
                     avatar.SetIKPosition(AvatarIKGoal.LeftFoot, hit.point +
footLoffset);
                     //Set the new rotation of IK
                     avatar.SetIKRotation(AvatarIKGoal.LeftFoot,
Quaternion.FromToRotation(transform.up, hit.normal) * transform.rotation);
                     //Save the collision position
                     footPosL = hit.point;
              }
              //Get the current position of the right foot
              footPosR = avatar.GetIKPosition(AvatarIKGoal.RightFoot);
        //[RayCast] to the ground, to know the distance
              if(Physics.Raycast(footPosR + Vector3.up, Vector3.down, out hit, 2.0f,
rayLayer))
              {
                     //Show [Ray]
                     Debug.DrawLine(hit.point, hit.point + hit.normal, Color.green);
                     //Set the new position of IK
                     avatar.SetIKPosition(AvatarIKGoal.RightFoot,hit.point +
footRoffset);
                     //Set the new rotation of IK
                     avatar.SetIKRotation(AvatarIKGoal.RightFoot,
Quaternion.FromToRotation(transform.up, hit.normal) * transform.rotation);
                     //Save the collision position
                     footPosR = hit.point;
              }
       }
       void WalkRunIK(){
              //Create this value to use the [RaycastHit]
              RaycastHit hit;
              Vector3 offsetVec = transform.forward * rayOffset;
        //Get the current position of the left foot
        footPosL = avatar.GetIKPosition(AvatarIKGoal.LeftFoot);
              //[RayCast] to the ground, to know the distance
              if(Physics.Raycast(footPosL + Vector3.up + offsetVec, Vector3.down, out
hit, 2.0f, rayLayer))
              {
                     //Show [Ray]
                     Debug.DrawLine(hit.point, hit.point + hit.normal, Color.yellow);
                     //Set the new position of IK
                     avatar.SetIKPosition(AvatarIKGoal.LeftFoot, hit.point + footLoffset
- offsetVec);
                     //Set the new rotation of IK
                     avatar.SetIKRotation(AvatarIKGoal.LeftFoot,
Quaternion.FromToRotation(transform.up, hit.normal) * transform.rotation);
                     //Save the collision position
                     footPosL = hit.point;


              }
              //Get the current position of the right foot
              footPosR = avatar.GetIKPosition(AvatarIKGoal.RightFoot);
        //[RayCast] to the ground, to know the distance
        if (Physics.Raycast(footPosR + Vector3.up + offsetVec, Vector3.down, out hit,
2.0f, rayLayer))
              {
                     //Show [Ray]
                     Debug.DrawLine(hit.point, hit.point + hit.normal, Color.green);
                     //Set the new position of IK
                     avatar.SetIKPosition(AvatarIKGoal.RightFoot,hit.point + footRoffset
- offsetVec);
                     //Set the new rotation of IK
                     avatar.SetIKRotation(AvatarIKGoal.RightFoot,
Quaternion.FromToRotation(transform.up, hit.normal) * transform.rotation);
                     //Save the collision position
                     footPosR = hit.point;
              }
       }
       void Update ()
       {
              //If [ikActive] is [true]
              if(ikActive){
                     //If the situation of the player is [Idle] and [ikActive] is [true]
                     if(avatar.GetBool("OnGround") == false)
            {
                            IdleUpdateCollider();
                     }
                     //If the situation of the player is [Walk] or [Run]
                     else if(avatar.GetBool("OnGround") == true)
            {
                            WalkRunUpdateCollider();
                     }
              //If [ikActive] is not [true]
              }else{
                     //Reset the values of my Collider
                     myCollider.center = new Vector3(0, Mathf.Lerp(myCollider.center.y,
defCenter.y, Time.deltaTime * smooth) ,0);
                     myCollider.height = Mathf.Lerp(myCollider.height, defHeight,
Time.deltaTime * smooth);
              }
```

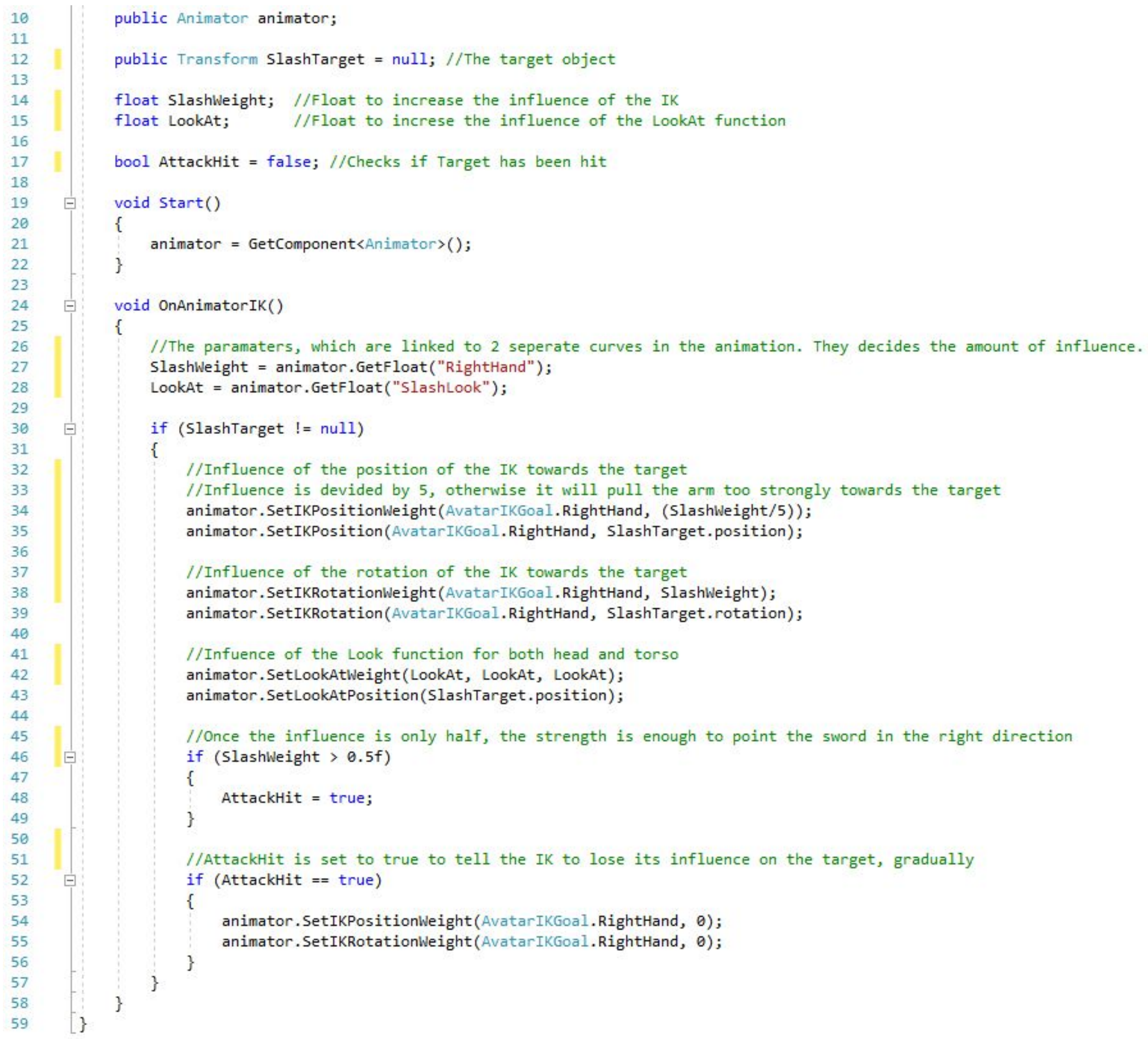
```
	}
	void IdleUpdateCollider ()
	{
		//Create this value to calculate the height difference of the feet
		float dif;
		//Calculate the height difference of the feet
		dif = footPosL.y - footPosR.y;
		//Do not let the value be less than 0
		if(dif < 0){dif *= -1;}
		//Change the Collider values depending on the value [dif]
		myCollider.center = new Vector3(0, Mathf.Lerp(myCollider.center.y, 
defCenter.y + dif, Time.deltaTime) ,0);
		myCollider.height = Mathf.Lerp(myCollider.height, defHeight - (dif / 2), 
Time.deltaTime);
	}
	void WalkRunUpdateCollider ()
	{
		//Create this value to use the [RaycastHit]
		RaycastHit hit;
		//Creating this value to save the height of the floor of the position I am
		Vector3 myGround = Vector3.zero;
		//Create this value to calculate the height difference
		Vector3 dif = Vector3.zero;
		//Check the height of the floor of the position I am
		if(Physics.Raycast(transform.position + Vector3.up, Vector3.down, out hit, 
3.0f, rayLayer))
		{
			//Save the value
			myGround = hit.point;
		}
		//RayCast to check the height of the position where I'm going
		if(Physics.Raycast(transform.position + (((transform.forward * 
(myCollider.radius))) + (myCollider.attachedRigidbody.velocity * Time.deltaTime)) + 
Vector3.up, Vector3.down, out hit, 2.0f, rayLayer))
		{
			//Show [Ray]
			Debug.DrawLine(transform.position + (((transform.forward * 
(myCollider.radius))) + (myCollider.attachedRigidbody.velocity * Time.deltaTime)) + 
Vector3.up, hit.point, Color.red);
			//Calculate the height difference between the height of the position 
I'm with the height of the position where I'm going
			dif = hit.point - myGround;
			//Do not let the value be less than 0
			if(dif.y < 0){dif *= -1;}
		}
		//If the [dif] is less than 0.5
		if(dif.y < 0.5f){
			//Change the Collider values depending on the value [dif]
			myCollider.center = new Vector3(0, Mathf.Lerp(myCollider.center.y, 
defCenter.y + dif.y, Time.deltaTime * smooth) ,0);
			myCollider.height = Mathf.Lerp(myCollider.height, defHeight - (dif.y 
/ 2), Time.deltaTime * smooth);
		//If the [dif] is not less than 0.5
		}else{
			//Reset the values of my Collider
			myCollider.center = new Vector3(0, Mathf.Lerp(myCollider.center.y, 
defCenter.y, Time.deltaTime * smooth) ,0);
```

```
			myCollider.height = Mathf.Lerp(myCollider.height, defHeight, 
Time.deltaTime * smooth);
		}

	}
}
```

14.2.5. Punch Combo IK

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

public class PunchCombo : MonoBehaviour {

    Animator anim;

    public bool ikActive = false;

    float rHWeight;
    float lHWeight;
    public float correctiveWeight = 1;

    public Transform[] Targets;
    private Transform currentTarget = null;
    private Transform prevTarget = null;
    public float Weight = 1;
    public float bodyWeight = 1;
    public float headWeight = 1;
    private int punchCounter = 0;
    private float journeyLength;
    private int prevRnd = 0;

    // Use this for initialization
    void Start () {
        anim = GetComponent<Animator>();
        currentTarget = getTarget();
    }

    // Update is called once per frame
    void Update()
    {
        punchCounter = anim.GetInteger("PunchCounter");
        Debug.Log(anim.GetInteger("PunchCounter"));
        Debug.Log("punchCounter: " + punchCounter);
        //updateWeights();
        if (punchCounter != 0 && punchCounter % 2 == 0)
        {
            resetTargets();
        }
    }

    private void resetTargets()
    {
        prevTarget = currentTarget;
        currentTarget = getTarget();
        //Debug.Log("targ: " + currentTarget.gameObject.name);
        journeyLength = Vector3.Distance(prevTarget.position, currentTarget.position);
        //Debug.Log("journeyLength: " + journeyLength);
    }

    private Transform getTarget()
    {
        int rnd = Random.Range(0, Targets.Length - 1); // pick a new one from 0 to 6
        while(rnd == prevRnd) // compare 0 vs. 0
        {
            rnd = Random.Range(0, Targets.Length - 1);
        }
        prevRnd = rnd;
        return Targets[rnd];
    }

    private void updateWeights()
    {
        //bodyWeight

        //headWeight

    }

    void OnAnimatorIK()
    {
        if (ikActive)
        {
            rHWeight = anim.GetFloat("RightHand");
            lHWeight = anim.GetFloat("LeftHand");


            anim.SetIKPosition(AvatarIKGoal.RightHand, currentTarget.position);
            anim.SetIKPositionWeight(AvatarIKGoal.RightHand, rHWeight /
correctiveWeight);

            anim.SetIKPosition(AvatarIKGoal.LeftHand, currentTarget.position);
            anim.SetIKPositionWeight(AvatarIKGoal.LeftHand, lHWeight / correctiveWeight);

            anim.SetLookAtWeight(Weight, bodyWeight, headWeight);
            anim.SetLookAtPosition(currentTarget.position);

        }
        else
        {
            anim.SetLookAtWeight(0);
            anim.SetIKPositionWeight(AvatarIKGoal.RightHand, 0);
            anim.SetIKPositionWeight(AvatarIKGoal.LeftHand, 0);

        }
    }
}
```

14.2.6. Slashing IK



```csharp
    public Animator animator;

    public Transform SlashTarget = null; //The target object

    float SlashWeight;  //Float to increase the influence of the IK
    float LookAt;       //Float to increse the influence of the LookAt function

    bool AttackHit = false; //Checks if Target has been hit

    void Start()
    {
        animator = GetComponent<Animator>();
    }

    void OnAnimatorIK()
    {
        //The paramaters, which are linked to 2 seperate curves in the animation. They decides the amount of influence.
        SlashWeight = animator.GetFloat("RightHand");
        LookAt = animator.GetFloat("SlashLook");

        if (SlashTarget != null)
        {
            //Influence of the position of the IK towards the target
            //Influence is devided by 5, otherwise it will pull the arm too strongly towards the target
            animator.SetIKPositionWeight(AvatarIKGoal.RightHand, (SlashWeight/5));
            animator.SetIKPosition(AvatarIKGoal.RightHand, SlashTarget.position);

            //Influence of the rotation of the IK towards the target
            animator.SetIKRotationWeight(AvatarIKGoal.RightHand, SlashWeight);
            animator.SetIKRotation(AvatarIKGoal.RightHand, SlashTarget.rotation);

            //Infuence of the Look function for both head and torso
            animator.SetLookAtWeight(LookAt, LookAt, LookAt);
            animator.SetLookAtPosition(SlashTarget.position);

            //Once the influence is only half, the strength is enough to point the sword in the right direction
            if (SlashWeight > 0.5f)
            {
                AttackHit = true;
            }

            //AttackHit is set to true to tell the IK to lose its influence on the target, gradually
            if (AttackHit == true)
            {
                animator.SetIKPositionWeight(AvatarIKGoal.RightHand, 0);
                animator.SetIKRotationWeight(AvatarIKGoal.RightHand, 0);
            }
        }
    }
}
```

14.3. References

Two Bone IK. (n.d.). Retrieved from https://docs.unrealengine.com/en-us/Engine/Animation/NodeReference/SkeletalControls/TwoBoneIK

UE4 - Principle behind Inverse Kinematics. (2015, October 10). Retrieved from https://www.youtube.com/watch?v=gHwhTEMlXHU

U. (2018, July 12). Unite Berlin 2018 - An Introduction to CCD IK and How to use it. Retrieved from https://www.youtube.com/watch?v=MA1nT9RAF3k&t=1530s

Unity 5 Tutorial The Built-In IK System. (2015, May 28). Retrieved from https://www.youtube.com/watch?v=EggUxC5_lGE

D. (2016, March 02). Unity Character Controller: Update #1 - IK Foot Placement, Ledge Climbing System. Retrieved from https://www.youtube.com/watch?v=NoGZ3I3lcbo&t=4s

N. (2017, May 16). Unity3d :Basic IK Tutorial. Retrieved from https://www.youtube.com/watch?v=6UgB7TMk3Bg&t=26s

D. (2014, August 23). Unreal Engine 4 - Inverse Kinematic (IK) feature. Retrieved from https://www.youtube.com/watch?v=KsJihbJCSDQ

https://forum.unity.com/threads/stuck-with-ik-any-good-step-by-step-inverse-kinematics-tutorials.415677/

Technologies, U. (n.d.). Inverse Kinematics. Retrieved from https://docs.unity3d.com/Manual/InverseKinematics.html

R. (2016, November 02). Tentacle Inverse Kinematics - VR - Unreal Engine 4. Retrieved from https://www.youtube.com/watch?v=MmrLJpW2c4g

The Best Animation Tricks of the Trade (For 2016). (2016, November 21). Retrieved from https://www.youtube.com/watch?v=_1j5Tf6ulII

H. (2017, October 19). Tutorial #1 || Active Ragdoll Unity || Edsonxn. Retrieved from https://www.youtube.com/watch?v=WbyScpgl5mw

T. (2014, August 26). Tutorial: Creating an FK/IK arm setup in Maya. Retrieved from https://www.youtube.com/watch?v=M6ViCN_sPVE

T. (2014, July 29). Tutorial: Rigging an IK Arm in Maya. Retrieved from https://www.youtube.com/watch?v=WkKx9ijydjk&t=63sT. (2014, July 22). Tutorial: Rigging an IK Spline Back in Maya. Retrieved from https://www.youtube.com/watch?v=Pj6TrmxhPew

Active ragdoll with authentic balance. (2016, August 07). Retrieved from https://www.youtube.com/watch?v=L8R3ta7KA5Q

C. (2016, February 12). Active Ragdolls in Unity 5. Retrieved from https://www.youtube.com/watch?v=GeJ1ZTQIWUI

Advanced foot IK for Unreal Engine 4 - (100% Free). (2017, May 04). Retrieved from https://www.youtube.com/watch?v=XetC9ivIXFc

Alan Zucconi. (2018, April 30). Inverse Kinematics for Tentacles. Retrieved from https://www.alanzucconi.com/2017/04/12/tentacles/

Animation Bootcamp: An Indie Approach to Procedural Animation. (2017, October 21). Retrieved from https://www.youtube.com/watch?v=LNidsMesxSE&t=784s

Augmented Reality. (n.d.). Retrieved from http://aboutaugmentedreality.blogspot.com/p/limitations-of-ar.html

Augmented Reality Tutorial No. 14: Augmented Reality using Unity3D and Vuforia (part 1). (2015, May 21). Retrieved from https://www.youtube.com/watch?v=qfxqfdtxyVA

Basic Switch IK/FK - Maya. (2016, October 16). Retrieved from https://www.youtube.com/watch?v=rGplirUI9R4

C. (2015, February 04). Character Rigging - Step 6 - IK/FK Arm (Autodesk Maya). Retrieved from https://www.youtube.com/watch?v=Qtua3919Gm0

S. (2017, March 17). Coding Challenge #64.2: Inverse Kinematics. Retrieved from https://www.youtube.com/watch?v=hbgDqyy8bIw

M. (2013, July 16). Creating a Character Rig - Part 10: Basic IK, FK, and result leg joints. Retrieved from https://www.youtube.com/watch?v=wDy6GQjPpp0

M. (2013, May 28). Creating a Character Rig - Part 5: Torso squash and stretch. Retrieved from https://www.youtube.com/watch?v=Ip-5PD3aNIg

Bereznyak, A. (2018, January 19). IK Rig: Procedural Pose Animation (GDC, Ed.). Retrieved from https://www.youtube.com/watch?v=KLjTU0yKS00&t=2330s

M. (2013, May 28). Creating a Character Rig - Part 6: Torso global transform and cleanup. Retrieved from https://www.youtube.com/watch?v=GTwe4Ejkn6Y&index=6&list=PL8hZ6hQCGHMXKqaX9Og4Ow52jsU_Y5veH

How to do inverse kinematics (IK) in Unity? (n.d.). Retrieved from https://answers.unity.com/questions/11106/how-to-do-inverse-kinematics-ik-in-unity.html

IK Setups. (n.d.). Retrieved from https://docs.unrealengine.com/en-us/Engine/Animation/IKSetups

IKinema RunTime and Unreal® Engine 4. (n.d.). Retrieved from https://www.ikinema.com/index.php?mod=documentation&show=184&id=206

Inverse Kinematics simple example #Unity3d. (2017, July 02). Retrieved from https://www.youtube.com/watch?v=GYfeALySSq8

Martindale, J. (2017, November 21). Headsets are just the beginning. How to make a VR rig for all your senses. Retrieved from https://www.digitaltrends.com/virtual-reality/what-you-need-for-full-vr-immersion/

Maya fk ik switch or ik fk switch. (2017, October 10). Retrieved from https://www.youtube.com/watch?v=TxI5DvcIdHE

Murray, J. H. (1997). Hamlet on the holodeck: The future of narrative in cyberspace. Cambridge, MA: The MIT press.

Rig an Ik Fk Leg - Maya (Part1). (2015, October 04). Retrieved from https://www.youtube.com/watch?v=tkQ95mGOj-U

H. (2017, August 15). RTIK Unreal Engine 4 inverse Kinematics Demo. Retrieved from https://www.youtube.com/watch?v=Cm-hjahqLh8

3. (2012, June 28). Spline iK Setup and Use. Retrieved from https://www.youtube.com/watch?v=rGRNJKQVIW4

Stuck with IK -- any good step by step Inverse Kinematics tutorials? (n.d.). Retrieved from https://forum.unity.com/threads/stuck-with-ik-any-good-step-by-step-inverse-kinematics-tutorials.415677/

Stapleton, D. (2017, March 20). Mass Effect: Andromeda Review. Retrieved from https://www.ign.com/articles/2017/03/20/mass-effect-andromeda-review

Martindale, J. (2018, December 17). Oculus Rift vs. HTC Vive: Prices drop, but our favorite stays the same. Retrieved from https://www.digitaltrends.com/virtual-reality/oculus-rift-vs-htc-vive/

Engine, U. (n.d.). Blueprint Best Practices. Retrieved from https://docs.unrealengine.com/en-us/Engine/Blueprints/BestPractices

Documentation, Unity scripting languages and you – Unity Blog. (n.d.). Retrieved from https://blogs.unity3d.com/2014/09/03/documentation-unity-scripting-languages-and-you/

AiGameDev.com. (2015, June 10). Nucl.ai Conference: Ubisoft Toronto "IK Rig" Prototype. Retrieved from https://www.youtube.com/watch?v=V4TQSeUpH3Q