Research and Developpement, Enjmin, 2020, August, 24

Inverse Kinematics for foot placement in Unity

Théo Olivier
Master's degree « Jeux et médias interactifs numériques », Angoulême, France
Coentitled by the Conservatoire national des arts et métiers and the University of Poitiers
Email: theo-olivier@laposte.net

Abstract—This document presents the results of my experimentation with using inverse kinematics for foot placement in 3D, with Unity and its integrated inverse kinematics system.

Keywords— Inverse Kinematics, Foot Placement, Unity, C#, Rigidbody Controller, 3D

I Introduction

When making a 3D controller, using animations (forward kinematics) is often enough to integrate the character in the game world, and make the player believe that his character is directly interacting with this world. But one thing that is pretty visible and can instantly break the immersion, is when the point where the character interacts with the world goes through solid surfaces.
In most cases not handled by the colliders, it is either the hands (because the character tries to grab something), or the foot (when the ground is not exactly as the animation artist imagined, most often a horizontal plane).
I choose to focus on the foot placement, using the inverse kinematics (IK) to correct the animations at runtime.

In this document, I'll summarize my experiment on this foot placement with IK. I'll first talk about the character controller I built. Then I'll discuss the hypotheses I tested with IK. And finally I'll present some possible solutions that could further upgrade the results on stairs.

II Character Controller

A *Movement*

The character has a Rigidbody and CapsuleCollider (the feet stick out an little, to be able to adjust and stick them to the ground).
I used the Input System Package (new input system) for the inputs, and the Cinemachine's Freelook Camera for a Third Person Sight camera.
To move the character, the rigidbody speed is modified using the inputs (left joystick with a gamepad). And the the last movement direction determines the character's direction.
The camera direction is controlled independently by an other input (right joystick).

B *Animator*

The character model used in my experiment is a bodyguard (human) from Batewar's "Bodyguards" Unity Asset Store's package. I used three animations, all from mixamo.com : an idle animation, a walking animation, and a stair climbing animation. I also wanted to use the descending stairs animation, but it was broken, and the reasoning is similar to the climbing one.

The character controller (CC) hands the normalized speed to the animator, wich is used to transition between the idle animation and the others.
A boolean parameter, changed by triggers in the scene, determines if the character is on stairs and transition between the climbing stairs animation and the walking one, and choose the fitting one right away when exiting the idle animation.
And all animations are made in place.

C *Step*

The CC as described above works well for movement on a plane or gradual or bumpy ground. But if there is a step, or stairs, the character is blocked.
Hence in the OnCollisionEnter method, if a step is detected, the character's position is moved towards the contact point, and the velocity is adjusted to the character's transform.up vector.
To detect if a collision is due to a step, we check if the difference between the character's collider lower point altitude and the collision altitude is less than the chosen max step height.
It is also important to check if this difference is greater than a chosen value (approximately one tenth of the character's height) to avoid a yo-yo effect when falling (even the fall from a single step).

III Inverse Kinematics

A Setup

For the IK to work properly, some preparation is needed.
First, the animator controller's layer's (I just used the "Base Layer") IK Pass parameter should be checked.
Then, the character model should be correctly rigged, and an avatar should be defined. The animation type could be Generic, but I chose Humanoid, since it is indeed the case for me, and it simplifies some things. The avatar should also be linked in the animator, and each animation (if the animations are in separate files from the model).
Also, as said in part II - A, the feet should stick out an little from the CapsuleCollider, to be able to adjust and stick them to the ground).
Finally, the IK related functions should be called from appropriate methods, like OnAnimationIK (MonoBehaviour) or OnStateIK (StateMachineBehaviour). I chose the first option.

To use Unity IK system (Unity (2020a)), it is possible to call the SetIKPosition and SetIKRotation methods (Animator). These set a target position and rotation respectively (in world space), that Unity will try to match. It is also possible to add a call to SetIKPositionWeight and SetIKRotationWeight (Animator), setting weights Unity will use to interpolate between the animation position and rotations, and the IK targets.
It is also possible to consult the animation defined position and rotation respectively (in world space), using GetIKPosition and GetIKPosition (Animator).
These methods will need an AvatarIKGoal (goal). These are named the foot and hand (left or right individually), but actually they are respectively the ankle and the wrist.

B Ankle Algorithm

The fist algorithm I used was the one I found in b3agz (2019), a YouTube video.

Position First, cast a downward ray starting from a little above the ankle.
If the ray collides, the IK target is the collision point, raised by a experimental constant (depends on the model), since the IK target should indicate the ankle position, and not the sole of the foot.

Rotation For the rotation, I modified a little b3agz (2019) algorithm.
Using the Unity space convention, z is forward, x is rightward, and y is upward.
Lets define rotFlat the rotation of the foot in the world xOz plane, and zFlat the forward component vector of the foot in world xOz plane ($zFlat = rotFlat * Vector3.forward$).
The foot y vector is simply the normal of the ground at the raycast collision point.
The x vector is the cross product between zFlat and y.
The z vector is then the cross product between y and x.
Using the Quaternion.LookRotation method, and the z and y vectors, we now have the foot IK rotation.

This rotation works pretty well, except when there is a big curvature on the ground. It fixes the heel on the ground, but humans tend to fix the middle of the foot on it, or even the toes. I tried making a linear interpolation between our resulting rotation and rotFlat. It is better for some cases, but worsens others. This is the reason why I tried the second algorithm.

C Toes Algorithm

This algorithm isn't much different, except that it tries to affix the middle of the sole of the foot, instead of the ankle. But it is not initially provided by Unity, so it's necessary to get the toe base joint one way or another (by linking it in a serialized variable for exemple).
The toe base is a joint, directly childed to the foot joint (i.e. the ankle), located approximately in the middle of the foot (in every axis).

First we keep the toe base position and the ankle to toe base vector in variables.
Then we do the ray cast, but from above the toe base position this time.
We calculate the rotation as previously.

Now what we have is the target for the middle of the foot, but the method needs the target for the ankle.
So first we take the collision point and raise it by a constant to get the toe base target (I added the vector computed from multiplying the ground normal - i.e. the foot up - by the constant to take the foot rotation into account).
Finally, to get the ankle position, we subtract the ankle to toe base vector that have been rotated by multiplying it by the target rotation and the inverse of the current rotation.

This method works way better, and is usable on flat horizontal ground, slopes, bumps, irregular ground, step, ... as long as the ground deformation is not too big in a little distance.

D Weights

In both algorithms, the weights should be set at 0 when the the foot is in the air, and at 1 when the foot is on the ground, with a smooth transition between the two.

To do that, first we create a float parameter in the animator for each IK goal (two since we only modify the feet).
Then in the animation files, we create a curve for each parameter, with imperatively the same name as the parameter it should modify, and set the curve in tune with the animation.
Finally, in our IK method, we get the current value of the parameter corresponding to our current goal and set with SetIKPositionWeight and SetIKRotationWeight.

IV Stairs and Upgrades

A Stairs

To complicate my experiment, I tried doing stairs.

First I used stairs made of steps, the trivial solution so to speak. All in all, it is not too bad, but the camera is shaking and the feet sometimes glitch a little.

So I tried doing fake stairs. The thing is physically a slope, but visually it looks like stairs. For that, I just put a slope in the steps, with the same angle as the stairs. Then I put the steps on a different layer from the slope, and disabled the collision between the character capsule collider and the steps (but not with the slope), and set the ray casts from our IK methods to collide with the steps.
The result is better, but the feet still slide a little.

B Upgrades

Since this was my research subject, I looked into how some triple A games do, and most use the simplest solution: they don't allow the player to see the feet from the side (by limiting the camera to the top of the character, or by syncing the character forward direction with the world xOz component vector of the forward of the camera). Indeed, even with the base animation without IK, by viewing the feet from the top or from behind, it is really hard to see the feet enter into the ground if just a bit.

If using our toes algorithm and fake stairs, if we could just stop the sliding, it should be near perfect. I thought of three possible solutions.
The first that I tested was to calculate the target position just once each time the foot touches the ground, but since the body still slid, the foot was left behind and it was not pretty to look at.
The second solution is in fact another cheat. I saw some games using stairs with steps so deep (in the z axis) that the character just walks several strides before climbing another step. So it's like having just a step alone, and our toes algorithm works pretty well for that, since it is really hard to see that the body and the foot slide normally.
The last solution, that I couldn't test, is to stop the sliding. For that, we could use Root Motion (Unity (2020b)) and Target Matching (Unity (2020c)). Root Motion is when we do not modify the speed directly, but it is the animation itself that moves the character. It allows the character to have a movement with non linear speed, and also perfect foot movement. The Target Matching modify animations to place the character in a target position at a certain moment of the animation (in the future).

V Conclusion

As discussed in this document, by using root motion, and displacing the ray cast from the ankle to the toe base, it is should be possible to have a pretty good foot placement with most surfaces and improve game immersion.
But it takes time to do, and hard to see normally, so in most projects, the game designers and programmers just position the camera so that the feet can't be seen on the side with a close view, saving resources and time for more core elements.

References

b3agz. 2019. REALISTIC Foot Placement Using IK in Unity. https://youtu.be/rGB1ipH6DrM.

Unity 2020a. *Inverse Kinematics*. Unity. https://docs.unity3d.com/Manual/InverseKinematics.html.

Unity 2020b. *Root Motion - how it works*. Unity. https://docs.unity3d.com/Manual/RootMotion.html.

Unity 2020c. *Target Matching*. Unity. https://docs.unity3d.com/Manual/TargetMatching.html.