

## Docs - Homebrew Foot IK (v1.1)



### Welcome, traveler!

Here, the core concepts of the HomebrewIK module are explained in depth.

You are free, and even encouraged to learn & modify the module and include it in your game. That's why I spent my time on writing a whole documentation page.

However, secondary distribution of the code is not allowed in default by the Standard Unity Asset Store EULA, so let's just be careful about that :D

#### [Required Scene Settings](#)

#### [Key Features](#)

[Gizmos & Advanced Inspector Visualizer](#)

[Foot Positioning by SphereCast](#)

[Height Correction by Orientation](#)

#### [Limitations](#)

[Floor's Normal Vector is always considered as Vector3.up](#)

[Non-Universal Default Body Positioning Implementation](#)

[No Toes IK Implemented by Default](#)

[No Dynamic IK Weight Resolver Implemented by Default](#)

#### [Notable Code Behaviors](#)

[IK Position Handling](#)

[Orientation References](#)

[Projected Angles](#)

[IK Rotation Handling](#)

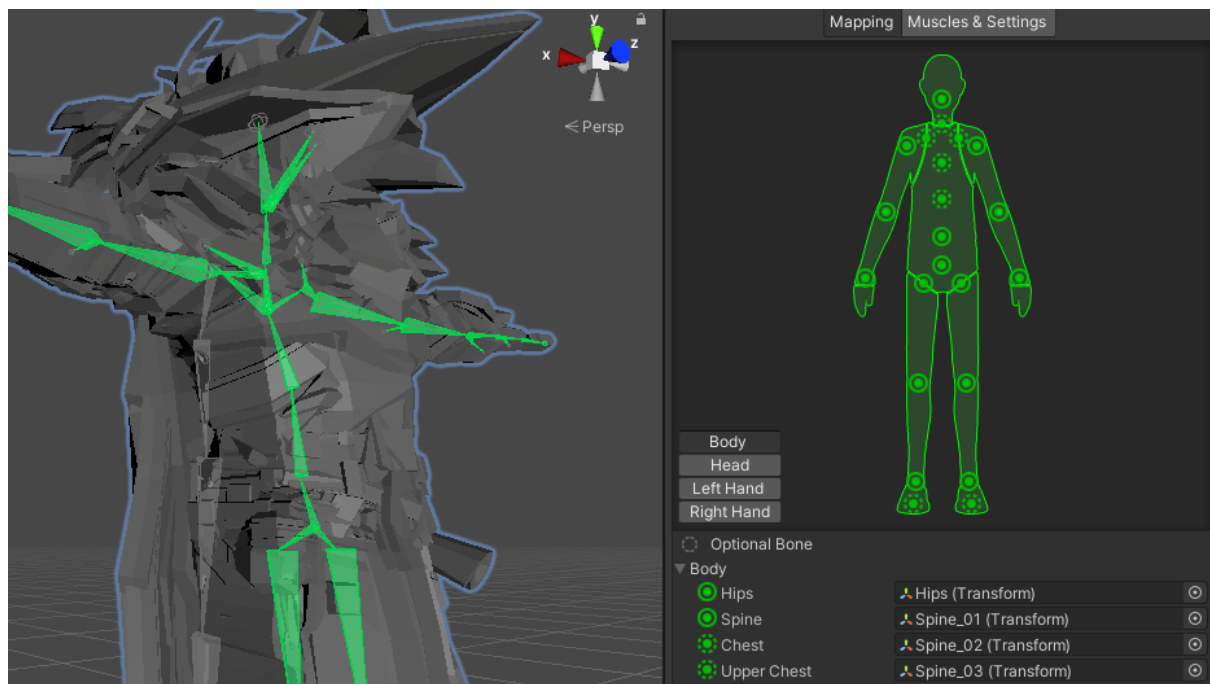
#### [License](#)

#### [Credits](#)

#### [Support](#)

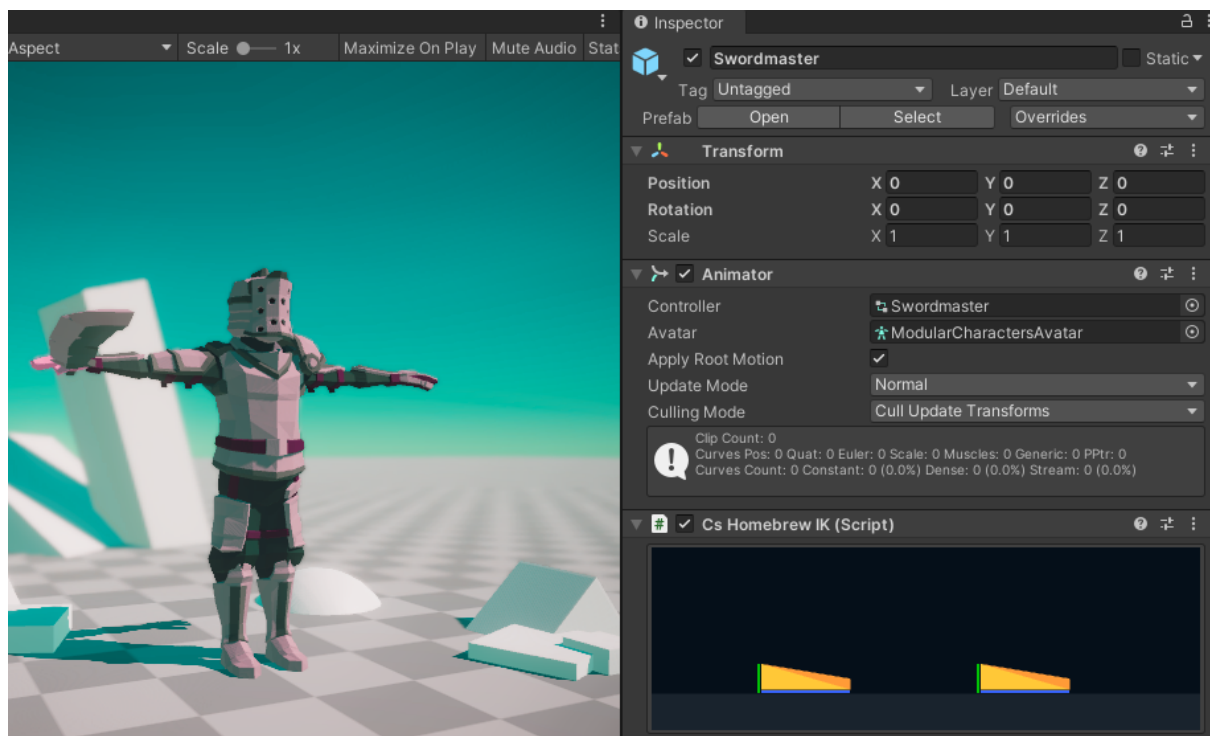
#### [Contact](#)

## Required Scene Settings



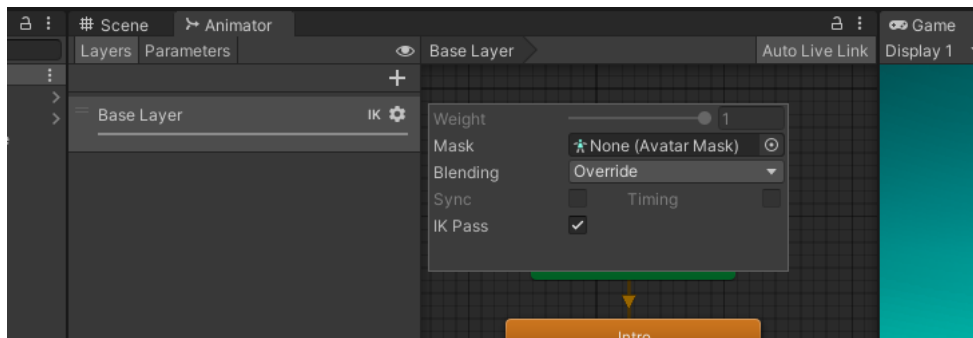
**A. The avatar that you are using must be rigged as humanoid to support Unity's IK features.**

Here, you may want to check if the toes bones are assigned to support animations that includes toes movement. One of the key feature is designed to work well even with toes animation.



**B. The script must be attached to the same gameobject that the animator is attached to.**

Or, you can modify the script's **Start()** function to fetch the animator differently. Since the IK logic relies heavily on animator interfaces, locating the right animator is really important.



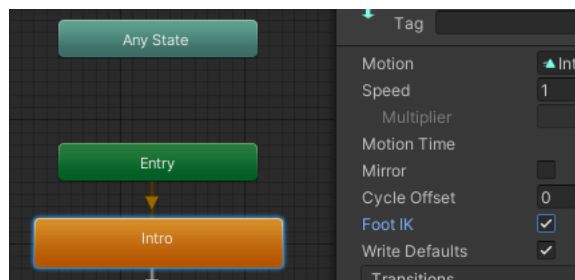
### C. IK Pass option inside the animator must be checked.

IK Pass option allows scripts to designate IK targets through the animator's interfaces.

Keep in mind that **setting an IK target is only effective per frame**, which means that IK related operations must be done every frame right after it gets resetted. For an example, getting and setting the same avatar bone's position by the same offset over and over will not change its position over time, but will only reposition the resetted value every frame before it gets rendered.

In short, calling IK property getters & setters will only be effective inside a limited scope. Think of it as a some sort of local variable. Below thread might help you understand this concept.

<https://forum.unity.com/threads/question-about-ik-targets.274011/>



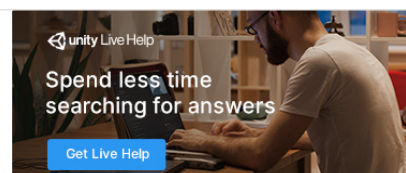
### D. (optional) Foot IK option for each state blocks may be toggled based on your preferences.

The **Foot IK** option which can be found inside each animator state blocks has nothing to do with dynamic raycasting and positioning. It's just a feature that matches bone positions of avatars having different dimensions to the one that the animation was created with, allowing users to **reuse the same animation for different avatars**.

what is the foot ik button for

i started using the ik in my project and wanted to use ik to make the character feet more realistically touch the ground. I started writing a script when i saw the foot ik button in the animator window. I look it up in the docs but i could not find out what exactly it does.

<https://answers.unity.com/questions/860198/what-is-the-foot-ik-button-for.html>



To elaborate this part for a bit, checking this option will match the foot bone's (which is equal to the ankle transform) transform position to the return value of **Animator.GetIKPosition()** function.

#### Animator.GetIKPosition

Suggest a change Thank you for helping us improve the quality of Unity Documentation. Although we cannot accept all submissions, we do read each suggested change from our users and will make updates where applicable. For some reason your suggested change could not be submitted. Please try again in a

<https://docs.unity3d.com/ScriptReference/Animator.GetIKPosition.html>



This function returns the so-called **IK goal** in world space. But remember, these goals will be reset every frame. So unless you set it afterwards using the **Animator.SetIKPosition()** function inside a specific frame, the return value will simply follow the animation curves.

The difference of this return value with the ankle transform's position is that without the **Foot IK** option toggled on, the ankle transform's position may not match the original animation's bone positions exactly, causing the foot to float or even dig underground.

For an example, it would be really hard for a pro baseball player to position all their bones to match an average person's bone positions, right? The same would be true for the opposite case, too.

Below is a result of a small experiment. Left image is off, and the right image is on. Small differences in values might be a floating point error, or code behaviors I think.

Left Foot Position	X	-0.2485	Y	0.08665	Z	1.45889
Left Foot IK Position	X	-0.2776	Y	0.06884	Z	1.46049

Left Foot Position	X	-0.2778	Y	0.06921	Z	1.45931
Left Foot IK Position	X	-0.2778	Y	0.06915	Z	1.45938

However, rotation returned by **Animator.GetIKRotation()** function will not be the same as the bone rotation even if the **Foot IK** option is on, so be careful about it.

## ☀ Key Features

### Gizmos & Advanced Inspector Visualizer

Gizmos will help you keep track of current ankle orientation, and various raycasting-related info. The button that allows you to toggle it is located at the upper side of the scene view.



The yellow dotted line shows you the **normal vector** of the ray hit point. The final IK rotation of each ankle will try to match the yellow ray, as long as it is in range of the **Max Rotation Angle** property.

The wired sphere floating above each ankle is the starting point of each ray, which is casted downwards in the world orientation (**Vector3.up** \* -1) by the **Ray Cast Range** property.

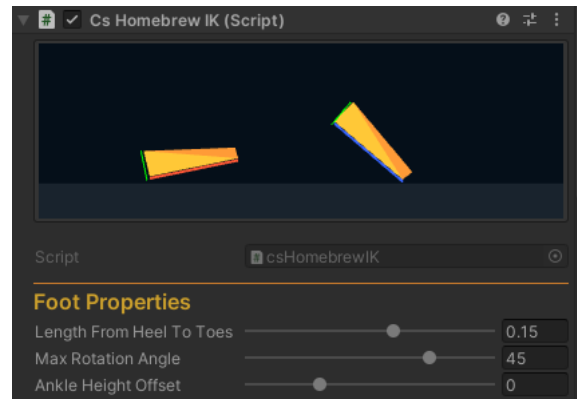
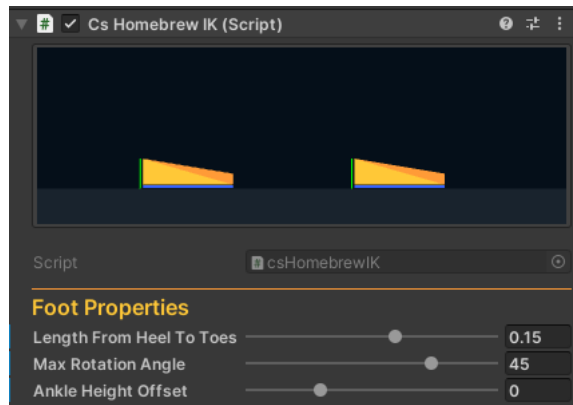


Height offset of these spheres can be adjusted inside the **Raycast Start Heights** property section.

Just in case that you would need to modify the implementation of this part to meet some special needs of yours, below is the part where the positions of those spheres are updated every frame.

csHomebrewIK.cs

```
341 private void UpdateRayHitInfo()
342 {
343     /* Rays will be casted from above each foot, in the downward orientation of the world */
344
345     leftFootRayStartPosition = leftFootTransform.position;
346     leftFootRayStartPosition.y += leftFootRayStartHeight;
347
348     rightFootRayStartPosition = rightFootTransform.position;
349     rightFootRayStartPosition.y += rightFootRayStartHeight;
```



Inside the inspector, a small display will help you intuitively understand how each property affects the module's behavior. Custom inspector-related code is located inside **editorHomebrewIK.cs** file.

## Foot Positioning by SphereCast

**Physics.Raycast()** will only return 'strictly' normal vectors, but I wanted it to smoothly slide down along both sharp and round edges. **Physics.SphereCast()** was chosen for this behavior.

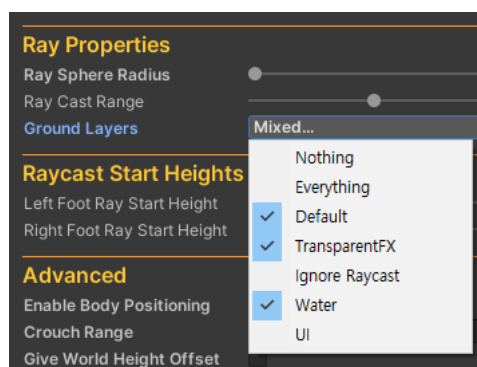
### Physics.SphereCast

Suggest a change Thank you for helping us improve the quality of Unity Documentation. Although we cannot accept all submissions, we do read each suggested change from our users and will make updates where applicable. For some reason your suggested change could not be submitted. Please try again in a

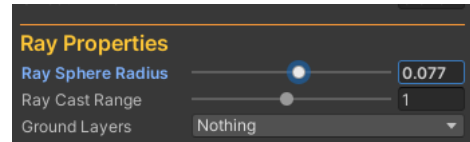
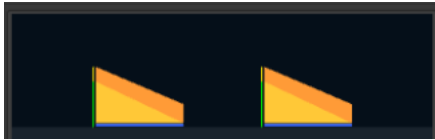
<https://docs.unity3d.com/ScriptReference/Physics.SphereCast.html>



To put it simple, **SphereCast()** is just a form of raycast that throws a giant ball at some direction, and tells you the impact point and the direction (normal vector) that it bounced off towards.

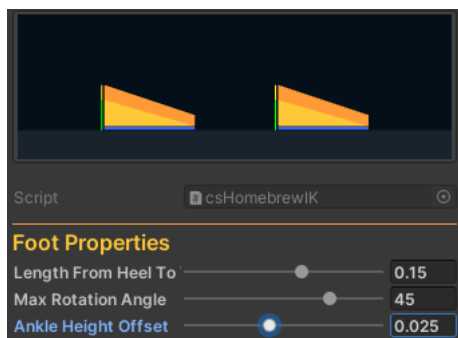


**Ground Layers** property will determine which layers the ray (sphere) can hit, and count it into the IK calculation. If nothing is checked, the script will assign the **"Default"** layermask on play.

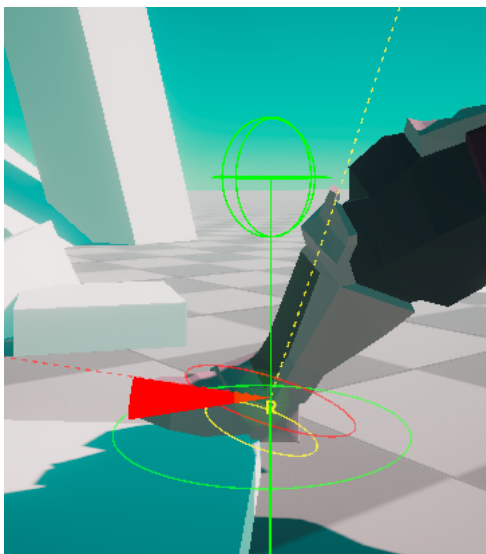


**Ray Sphere Radius** property will be used to determine each ankle's height. The length of each vertical green line inside the Inspector Visualizer will rely on this property. Remember, it is radius, not the diameter.

The reason for the sphere's radius itself being used as the height of the ankle is that the raycast returns the position of the center of the sphere, and all IK rotations will happen on that position. Setting the foot's height to be higher or lower than the radius would produce unwanted results. Therefore, the radius is the perfect value for sliding along sharp edges, following the normal vector returned by **SphereCast()** itself.



However, there may be situations where the avatar's ankle height may be higher than average, so that you want to give an offset. You can adjust the ankle's height with the **Ankle Height Offset** property. The length of this offset will be marked by a yellow line in the inspector visualizer.



Here, keep in mind that all stored vectors rely solely on **each ankle's transform**. Not the toes, nor the centroid of the foot mesh. (there's literally zero mesh-related operations in the script)

To be specific, the script will capture the **initial forward direction vector** of the parent transform, (the one that the script is attached to) and this vector will be rotated by the **delta rotation** (amount of change) of each ankle's transform. This, will be stored as the **final direction vector of each foot** and be utilized throughout various vector operations inside the script.

The reason for this is that we are fetching & setting the avatar **bones'** position and orientation for the IK, not the foot **mesh**. And where's that bone? At the same position as the ankle.

csHomeBrewIK.cs

```

309 //This is being done because we want to know in what angle did the foot go underground
310 참조 1개
311 private void UpdateFootProjection()
312 {
313     /* This is the only part in this script (except for those gizmos) that accesses footOrientationReference */
314     leftFootDirectionVector = leftFootOrientationReference.rotation * initialForwardVector;
315     rightFootDirectionVector = rightFootOrientationReference.rotation * initialForwardVector;

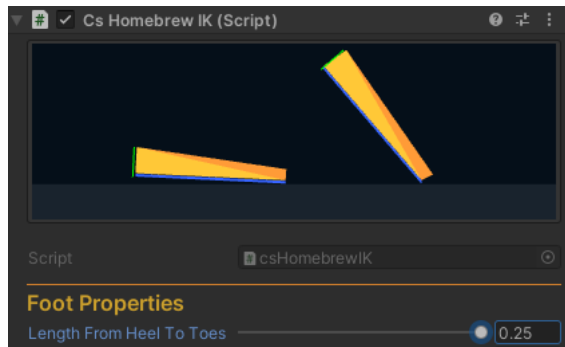
```

This means that manually configuring each avatar bones' orientation will affect the final visual **orientational offset** of each foot, but the actual IK operations will be performed just the same, as long as the position of ankles and the script properties are left unchanged.

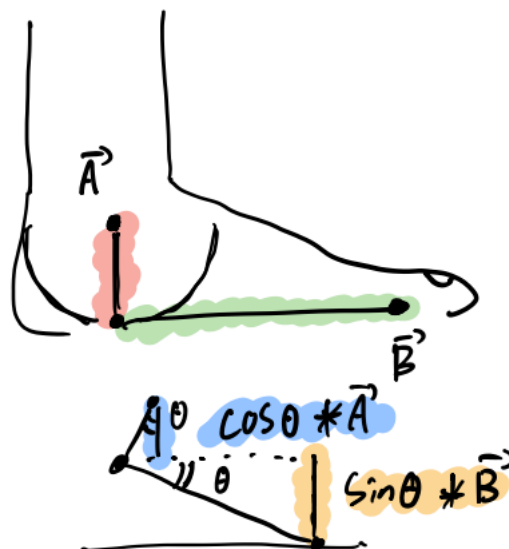
In other words, you can even make your character walk on high heels with this module.

## Height Correction by Orientation

Whenever an animation causes the foot to go underground, there will be a **height correction** made based on the foot's length & orientation. Without this, foot parts in front of the ankle will keep digging underground. (this was the reason that I wrote this IK at the first place)



**Length From Heel To Toes** property will be the base reference value for this operation. The longer the length is, the larger the height correction will be. It is recommended to match it to the actual foot meshes' dimensions. Below is a sketch that I drew while creating this module.

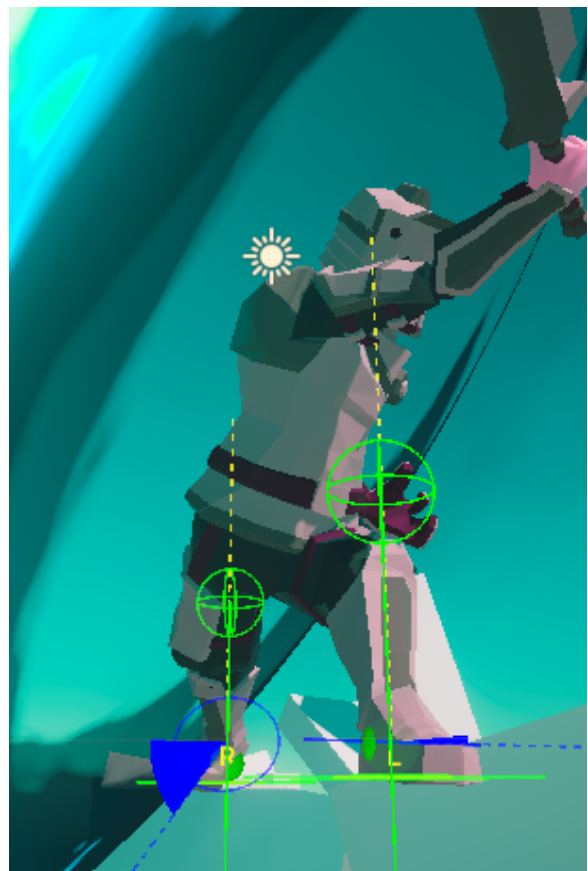
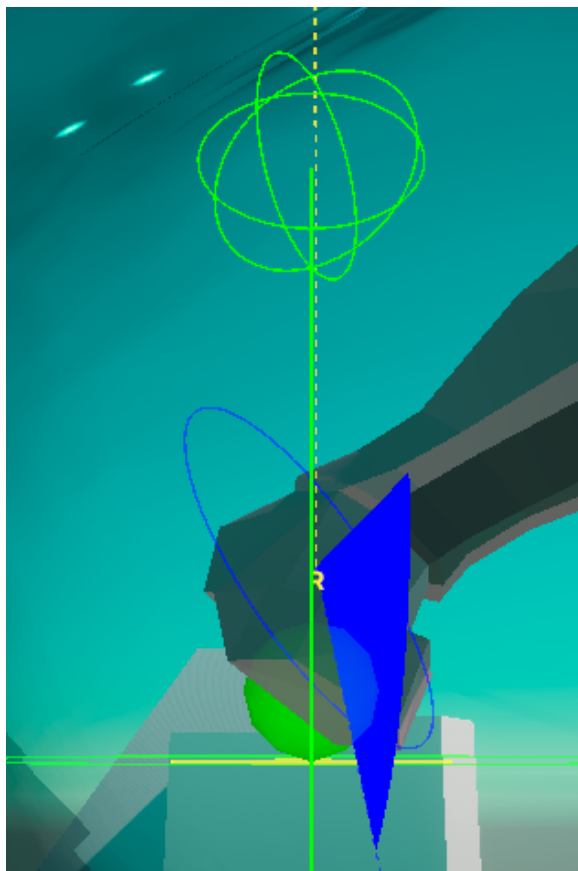


csHomeBrewIK.cs

```

419      /* Foot height correction based on the projected angle */
420
421      float trueLeftFootProjectedAngle = leftFootProjectedAngle - leftFootRayHitProjectedAngle;
422
423      if (trueLeftFootProjectedAngle > 0)
424      {
425          leftFootHeightOffset += Mathf.Abs(Mathf.Sin(
426              Mathf.Deg2Rad * trueLeftFootProjectedAngle) *
427              lengthFromHeelToToes);
428
429          // There's no Abs here to support negative manual height offset
430          leftFootHeightOffset += Mathf.Cos(
431              Mathf.Deg2Rad * trueLeftFootProjectedAngle) *
432              GetAnkleHeight();
433      }
434      else
435      {
436          leftFootHeightOffset += GetAnkleHeight();
437      }

```



**Note that this method expects the toes to be either barely noticeable (dimension-wise) or properly animated in straight orientation against the floor.**

## Limitations

### Floor's Normal Vector is always considered as Vector3.up

If you want the module to work even in extreme conditions like upside down, you should write a custom interface that returns the normal vector of the desired floor, and replace some parts of the code that relies on world vector constants for floor reference.



## Non-Universal Default Body Positioning Implementation

Different games utilize different techniques for handling representation of character transforms.

The default implementation just adds up the lower foot's position to the body positioning interface, and is not robust against all kinds of environments by any means. You may want to use raycasting, or even modify the parent transform's properties itself at some cases.

In these cases, you can just modify the body positioning function yourself.

## No Toes IK Implemented by Default

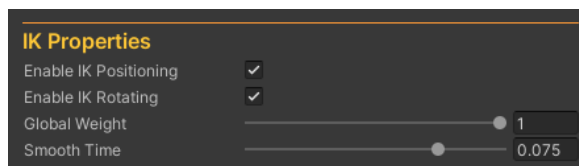
Implementing an IK system for foot is quite straightforward to be honest, but for toes, there are loads of possible ways of implementing it. It is a matter of preferences, so I will leave it to you.

Note that since the height correction logic only counts the length from heel to toes, you should not need to remove existing code blocks in order to additionally implement a Toes IK for yourself.

## No Dynamic IK Weight Resolver Implemented by Default

There may be some situations that you may want to deactivate the module for a sec, like jumping.

Likewise, the appropriate implementation largely depends on the game that you're developing, so I will leave this part to you as well.



Note that there's a **Global Weight** property that can be modulated with either script or animation clips. Properties are currently set to private, but you can define getters to deal with it. Or even create separate variables for each foot!

Also, keep in mind that the default implementation's movement smoothing part does nothing with the weight, it only smoothly lerps over two separate vectors. It is free for you to wrangle with.

## Notable Code Behaviors

### IK Position Handling

csHomeBrewIK.cs

```
참조 1개
532 private void ApplyFootIK()
533 {
534     /* Position Handling */
535
536     CopyByAxis(ref leftFootIKPositionBuffer, playerAnimator.GetIKPosition(AvatarIKGoal.LeftFoot),
537               true, false, true);
538
539     CopyByAxis(ref rightFootIKPositionBuffer, playerAnimator.GetIKPosition(AvatarIKGoal.RightFoot),
540               true, false, true);
```

IK position calculations are only applied for the y axis, and other axes are left untouched.

One of the reason that the above function calls are included in the **OnAnimatorIK()** function is that if the position to copy to the foot position is fetched inside the **Update()** function, the foot will not move an inch, because the final x and z values will be set

**before the animation curves are applied to the avatar bones.** The value that was set before the animation's curve application will override the movement caused by the animation later, causing an undesired behavior when implemented.

#### Order of execution for event functions

Running a Unity script executes a number of event functions in a predetermined order. This page describes those event functions and explains how they fit into the execution sequence. The diagram below summarizes how Unity orders and repeats event functions over a script's lifetime.

<https://docs.unity3d.com/Manual/ExecutionOrder.html>



- **OnAnimatorIK:** Sets up animation IK. This is called once for each Animator Controller layer with **IK pass** enabled.

This event executes only if you are using a Humanoid rig.

In my case, I have toggled on the Foot IK option inside the animator state inspector. So, calling the **GetIKPosition()** function and fetching the position of bone transform will return the same value, and work just the same. But for consistency & universal code behavior, I chose the IK interface.

## Orientation References

csHomeBrewIK.cs

```
295 | /* These gameobjects hold different orientation values from footTransform.rotation, but the delta remains the same */
296 |
297 | leftFootOrientationReference = new GameObject("[RUNTIME] Normal_Orientation_Reference").transform;
298 | rightFootOrientationReference = new GameObject("[RUNTIME] Normal_Orientation_Reference").transform;
299 |
300 | leftFootOrientationReference.position = leftFootTransform.position;
301 | rightFootOrientationReference.position = rightFootTransform.position;
302 |
303 | leftFootOrientationReference.SetParent(leftFootTransform);
304 | rightFootOrientationReference.SetParent(rightFootTransform);
```

The reason that I created new objects for tracking foot orientation is due to quaternions.

**Quaternion.eulerAngles** interface is practically unusable in these kind of applications, (try out the related article in the credits part) so we must deal with raw quaternions instead of vectors to deal with foot rotations.

However, as stated above, I wanted the **delta rotation of each ankle** in order to use it with the forward direction vector of the parent transform, and not the raw bone orientation of each ankle.

Creating an interface that stores the initial raw bones' orientation and applies the inverse of that rotation to current bone orientation to return the final quaternion that I want is certainly possible, but should be an unnecessary hassle.

Plus, imaging an empty child object is much more intuitive to understand than capturing and tracking down multiple rotation values.

## Projected Angles

csHomeBrewIK.cs

```

309 //This is being done because we want to know in what angle did the foot go underground
310 참조 1개 private void UpdateFootProjection()
311 {
312     /* This is the only part in this script (except for those gizmos) that accesses footOrientationReference */
313
314     leftFootDirectionVector = leftFootOrientationReference.rotation * initialForwardVector;
315     rightFootDirectionVector = rightFootOrientationReference.rotation * initialForwardVector;
316
317     /* World space based vector defines are used here for the representation of floor orientation */
318
319     leftFootProjectionVector = Vector3.ProjectOnPlane(leftFootDirectionVector, Vector3.up);
320     rightFootProjectionVector = Vector3.ProjectOnPlane(rightFootDirectionVector, Vector3.up);
321
322     /* Cross is done in this order because we want the underground angle to be positive */
323
324     leftFootProjectedAngle = Vector3.SignedAngle(
325         leftFootProjectionVector,
326         leftFootDirectionVector,
327         Vector3.Cross(leftFootDirectionVector, leftFootProjectionVector) *
328             // This is needed to cancel out the cross product's axis inverting behaviour
329             Mathf.Sign(leftFootDirectionVector.y));
330
331     rightFootProjectedAngle = Vector3.SignedAngle(
332         rightFootProjectionVector,
333         rightFootDirectionVector,
334         Vector3.Cross(rightFootDirectionVector, rightFootProjectionVector) *
335             // This is needed to cancel out the cross product's axis inverting behaviour
336             Mathf.Sign(rightFootDirectionVector.y));
337 }

```

Projected angles are needed in order to perform the foot height correction. They are calculated by literally projecting the vector to the floor plane, supposed to have a normal vector of **Vector3.up**.

Here, keep in mind that **all vector constants inside the script are chosen carefully between local and world vectors** to support any orientational offsets in parent transform. In other words, this module should work well even if you lean the parent transform in various directions.

These angles are positive when the toes go underground, and negative when the toes point upwards. Just a personal preference.

csHomeBrewIK.cs

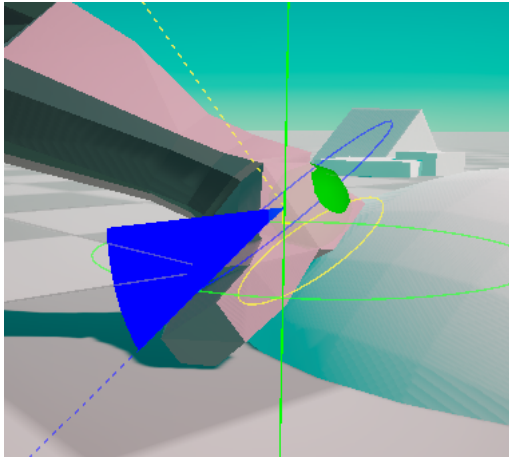
```

367 // Left Foot Ray Handling
368 if (leftFootRayHitInfo.collider != null)
369 {
370     leftFootRayHitHeight = leftFootRayHitInfo.point.y;
371
372     /* Angle from the floor is also calculated to isolate the rotation caused by the animation */
373
374     // We are doing this crazy operation because we only want to count rotations that are parallel to the foot
375     leftFootRayHitProjectionVector = Vector3.ProjectOnPlane(
376         leftFootRayHitInfo.normal,
377         Vector3.Cross(leftFootDirectionVector, leftFootProjectionVector));
378
379     leftFootRayHitProjectedAngle = Vector3.Angle(
380         leftFootRayHitProjectionVector,
381         Vector3.up);
382 }
383 else
384 {
385     leftFootRayHitHeight = transform.position.y;
386 }

```

Projected angles of ray hit normals are also needed in order to isolate the rotation caused by the animation from the rotation caused by normal vector returned from raycasting.

The cross product may seem a bit complex, but it just returns a normal vector that is perpendicular to the foot's forward direction vector.



If these two angles are not isolated from each other, the module will apply height correction even in situations like the above image, where there should not be any height correction being applied.

## IK Rotation Handling

csHomeBrewIK.cs

```

551      /* Rotation Handling */
552
553      /* This part may be a bit tricky to understand intuitively, refer to docs for an explanation in depth */
554
555      // FromToRotation is used because we need the delta, not the final target orientation
556      Quaternion leftFootRotation =
557          Quaternion.FromToRotation(transform.up, leftFootIKRotationBuffer) *
558          playerAnimator.GetIKRotation(AvatarIKGoal.LeftFoot);
559
560      // FromToRotation is used because we need the delta, not the final target orientation
561      Quaternion rightFootRotation =
562          Quaternion.FromToRotation(transform.up, rightFootIKRotationBuffer) *
563          playerAnimator.GetIKRotation(AvatarIKGoal.RightFoot);
564
565      playerAnimator.SetIKRotationWeight(AvatarIKGoal.LeftFoot, globalWeight);
566      playerAnimator.SetIKRotationWeight(AvatarIKGoal.RightFoot, globalWeight);
567
568      if (enableIKRotating == true)
569      {
570          playerAnimator.SetIKRotation(AvatarIKGoal.LeftFoot, leftFootRotation);
571          playerAnimator.SetIKRotation(AvatarIKGoal.RightFoot, rightFootRotation);
572      }
573  
```

At a glance, it may seem that this part of the code could introduce some problems by infinitely roating (multiplication between quaternions performs a rotation) the final IK rotation by getting and setting the same property over and over, but it is not.

Just like the IK position handler, rotation gets reset every frame also, allowing getting and setting the same value over and over without causing a change of value over time.

Sadly, the actual lowest level implementations of the animator interfaces could not be found online, so most of the learnings related to the interface behavior which is poorly documented by unity (including the things that I wrote on this page) only relies on trial & error for now...

UnityCsReference/Animator.bindings.cs at c84064be69f20dcf21ebe4a7bbc176d48e2f289c · Unity-Technologies/UnityCsReference  
 Unity C# reference source code. Contribute to Unity-Technologies/UnityCsReference development by creating an account on GitHub.

<https://github.com/Unity-Technologies/UnityCsReference/blob/c84064be69f20dcf21ebe4a7bbc176d48e2f289c/Modules/Animation/ScriptBinding/s/Animator.bindings.cs>

Unity-Tec  
**UnityCsR**  
 Unity C# reference

0  
 Contributors

## License

### Asset Store Terms of Service and EULA - Unity

The Unity Asset Store ("Unity Asset Store") is owned and operated by Unity Technologies ApS (company no. 30 71 99 13), Niels Hemmingsens Gade 24, 1153 Copenhagen K, Denmark ("Unity").  
[https://unity3d.com/legal/as\\_terms?\\_gl=1\\*wp1edc\\*\\_gcl\\_aw\\*R0NMLjE2NTAxMjQ3NzEuQ2owS0NRancwdW1TQmhEckFSSXNBSDDGQ29lLUdrS2l5YjI3aE5LYjM0LVBQUFZraGtieTU2TTBpUGlDSFNMMjhiSVpMN3IwUmttWk92b2FBdjN1RUFMd193Y0I](https://unity3d.com/legal/as_terms?_gl=1*wp1edc*_gcl_aw*R0NMLjE2NTAxMjQ3NzEuQ2owS0NRancwdW1TQmhEckFSSXNBSDDGQ29lLUdrS2l5YjI3aE5LYjM0LVBQUFZraGtieTU2TTBpUGlDSFNMMjhiSVpMN3IwUmttWk92b2FBdjN1RUFMd193Y0I)



## Credits

### Inverse Kinematics

Most animation is produced by rotating the angles of joints in a skeleton to predetermined values. The position of a child changes according to the rotation of its parent and so the end point of a chain of joints can be determined from the angles and relative positions of the individual joints it contains.

 <https://docs.unity3d.com/Manual/InverseKinematics.html>




### Euler Angles

When you started developing games in Unity, you were introduced to a term, most likely for the first time: "Quaternion". Putting in rotations in the Inspector makes sense: You type in three numbers for how you wanted to rotate this thing, and that was that. And then you start programming.

<https://starmanta.gitbooks.io/unitytipsredux/content/second-question.html>

### How to enable/disable a List in Unity inspector using a bool?


It is not possible to gray out the entire list without writing a custom editor window. In the Unity source code, you can see that attributes are never applied to arrays. In PropertyHandler.cs within the method HandleDrawnType, beginning on line 102: // Use PropertyDrawer on array elements, not on array itself.

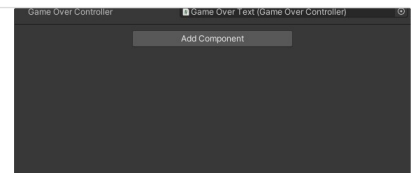
 <https://stackoverflow.com/questions/58441744/how-to-enable-disable-a-list-in-unity-inspector-using-a-bool>



### Making the Inspector Look Better

Unity recently had a sale on some of its assets and one of them was the Odin Inspector. I am a big fan of using already made assets to assist in my work flow, instead of reinventing the wheel. One of...

 <https://blog.devgenius.io/making-the-inspector-look-better-175baf39ada0>



## Support

[keithrek@hanmail.net](mailto:keithrek@hanmail.net)

## Contact

### SeungGeon Kim - Technical Game Designer (Project R.O.C.O) - REDJACK | LinkedIn

View SeungGeon Kim's profile on LinkedIn, the world's largest professional community. SeungGeon has 2 jobs listed on their profile. See the complete profile on LinkedIn and discover SeungGeon's connections and jobs at similar companies.

 <https://www.linkedin.com/in/keithrek/>

