User Manual

Welcome to Final IK.

IK Components

Final IK contains a number of powerful high speed IK components.

Aim

AimIK solver is a modification of the CCD algorithm that rotates a hierarchy of bones to make a child Transform of that hierarchy aim at a target. It differs from the basic built-in Animator.SetLookAtPosition or the LookAtIK functionality, because it is able to accurately aim transforms that are not aligned to the main axis of the hierarchy.

AimIK can produce very stabile and natural looking retargeting of character animation, it hence has great potential for use in weapon aiming systems. With AimIK we are able to offset a single forward aiming pose or animation to aim at targets even almost behind the character. It is only the Quaternion singularity point at 180 degrees offset, where the solver can not know which way to turn the spine. Just like LookAtIK, AimIK provides a clampWeight property to avoid problems with that singularity issue.

AimIK also works with rotation limits, however it is more prone to get jammed than other constrained solvers, should the chain be heavily constrained.

Aim provides high accuracy at a very good speed, still it is necessary to keep in mind to maintain the target position at a safe distance from the aiming Transform. If distance to the target position is less than distance to the aiming Transform, the solver will try to roll in on itself and might be unable to produce a finite result.

Getting started:

- Add the AimIK component to your character
- Assign the spine bones to "Bones" in the component
- Assign the Aim Transform (the Transform that you want to aim at the target)
- Make sure Axis represents the local axis of the Aim Transform that you want to be aimed at the target
- Set weight to 1, press Play

Changing the aiming target:

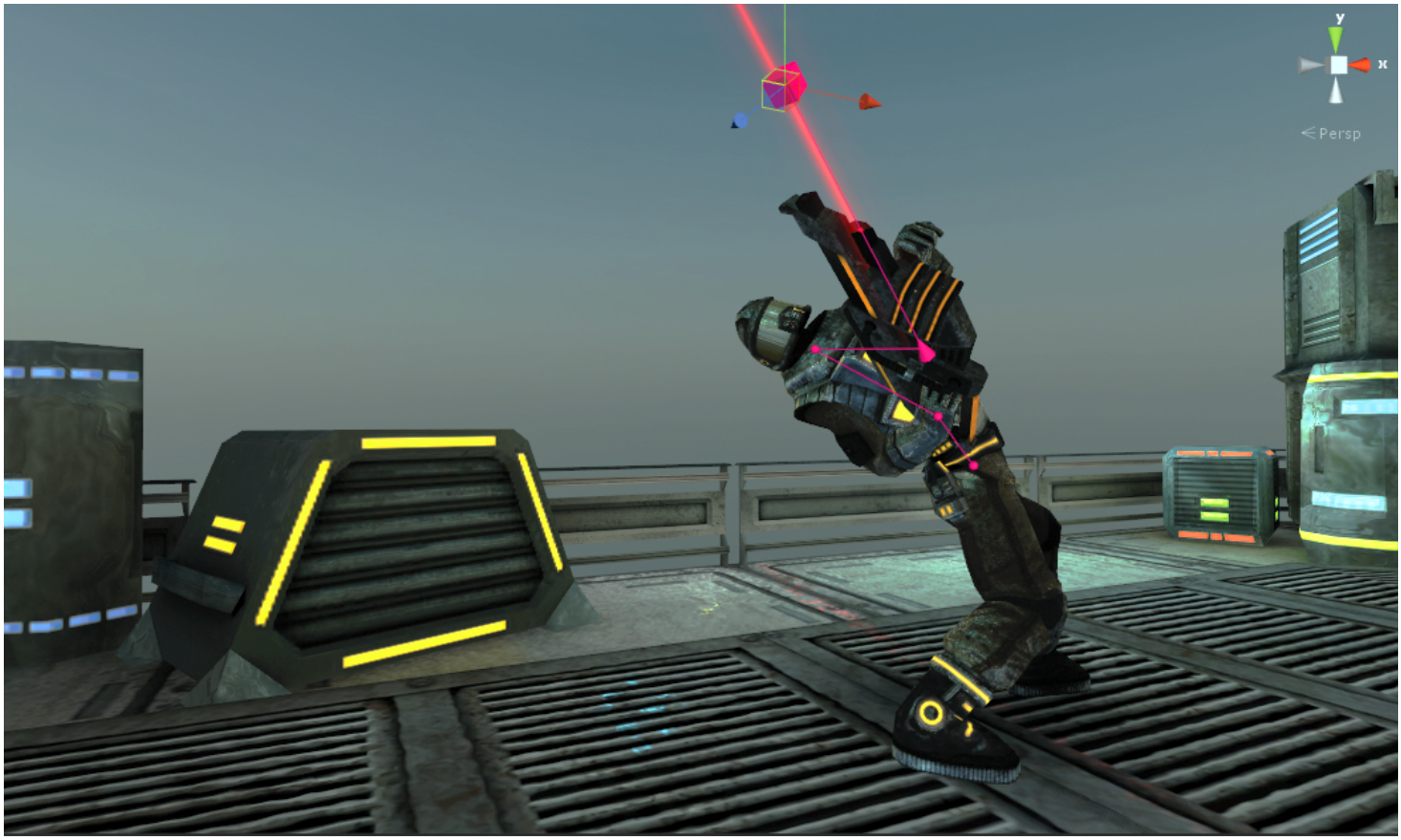
```
public AimIK aimIK;

void LateUpdate () {
    aimIK.solver.IKPosition = something;
}
```

Changing the Aim Transform:

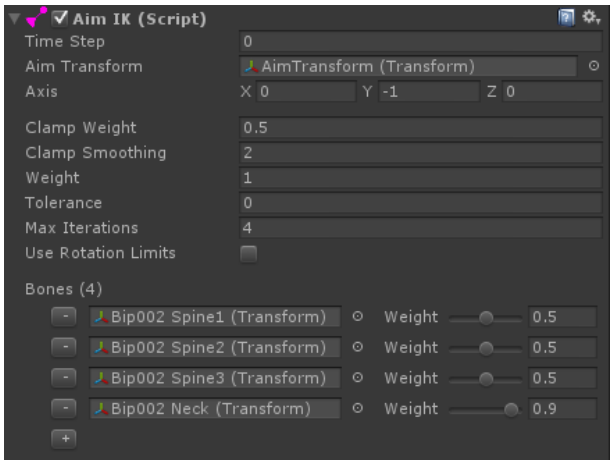
```
public AimIK aimIK;

void LateUpdate () {
    aimIK.solver.transform = something;
    aimIK.solver.axis = localAxisOfTheTransformToAimAtTheTarget;
}
```

Adding AimIK in runtime:

- Add the AimIK component via script
- Call AimIK.solver.SetChain()



The AimIK solver in action



The AimIK component

Biped IK

IK system for standard biped characters that is designed to replicate and enhance the behaviour of the Unity's built-in character IK setup.

BipedIK has many benefits over Unity's Animator IK.
Firstly, Animator IK does not allow the modifiaction of any of even the most basic solver parameters, such as limb bend direction, which makes the system difficult, if not impossible to use or extend in slightly more advanced use cases. Even in the simplest of cases, Animator can produce unnatural poses or bend a limb in unwanted direction and there is nothing that can be done to work around the problem.
Secondly, Animator IK lacks a spine solver.
Thirdly, Animator's LookAt functionality can often solve to weird poses such as bending the spine backwards when looking over the shoulder.
BipedIK also incorporates AimIK.
Last, but not least, BipedIK does NOT require Unity Pro.

To simplify migration from Unity's built-in Animator IK, BipedIK supports the same API, so you can just go from animator.SetIKPosition(...) to bipedIK.SetIKPosition(...).

BipedIK, like any other component in the FinalIK package, goes out of it's way to minimize the work required for set up. BipedIK automatically detects the biped bones based on the bone structure of the character and the most common naming conventions, so unless you have named your bones in Chinese, you should have BipedIK ready for work as soon as you can drop in the component. If BipedIK fails to recognize the bone references or you just want to change them, you can always manage the references from the inspector.

Getting started:

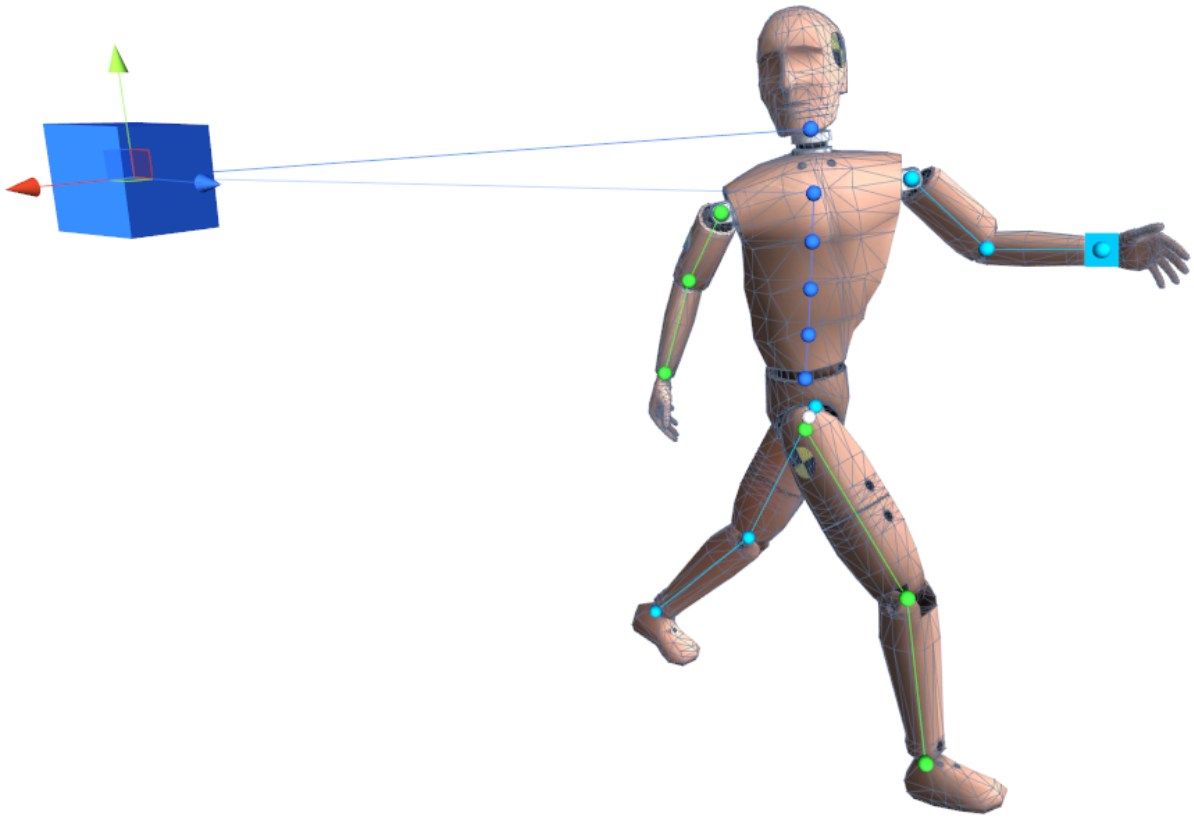
- Add the BipedIK component to the root of your character (the same GameObject that has the Animator/Animation component)
- Make sure the auto-detected biped references are correct
- Press play, weigh in the solvers

Accessing the solvers of Biped IK:

```
public BipedIK bipedIK;

void LateUpdate () {
    bipedIK.solvers.leftFoot.IKPosition = something;
    bipedIK.solvers.spine.IKPosition = something;
    ...
}
```

Adding BipedIK in runtime:

- Add the BipedIK component via script
- Assign BipedIK.references
- Optionally call BipedIK.SetToDefaults() to set the parameters of the solvers to default BipedIK values. Otherwise default values of each solver are used.



Testing BipedIK in the scene view

The BipedIK component

CCD

CCD (Cyclic Coordinate Descent) is one of the simplest and most popular inverse kinematics methods that has been extensively used in the computer games industry. The main idea behind the solver is to align one joint with the end effector and the target at a time, so that the last bone of the chain iteratively gets closer to the target.

CCD is very fast and reliable even with rotation limits applied. CCD tends to overemphasise the rotations of the bones closer to the target position. Reducing bone weight down the hierarchy will compensate for this effect. It is designed to handle serial chains, thus, it is difficult to extend to problems with multiple end effectors (in this case go with FABRIK). It also takes a lot of iterations to fully extend the chain.

Monitoring and validating the IK chain each frame would be expensive on the performance, therefore changing the bone hierarchy in runtime has to be followed by calling SetChain (Transform[] hierarchy) on the solver. SetChain returns true if the hierarchy is valid.

CCD allows for direct editing of it's bones' rotations (not by the scene view handles though), but not positions, meaning you can write a script that rotates the bones in a CCD chain each frame, but you should not try to change the bone positions like you can do with a FABRIK solver. You can, however, rescale the bones at will, CCD does not care about bone lengths.

Getting started:

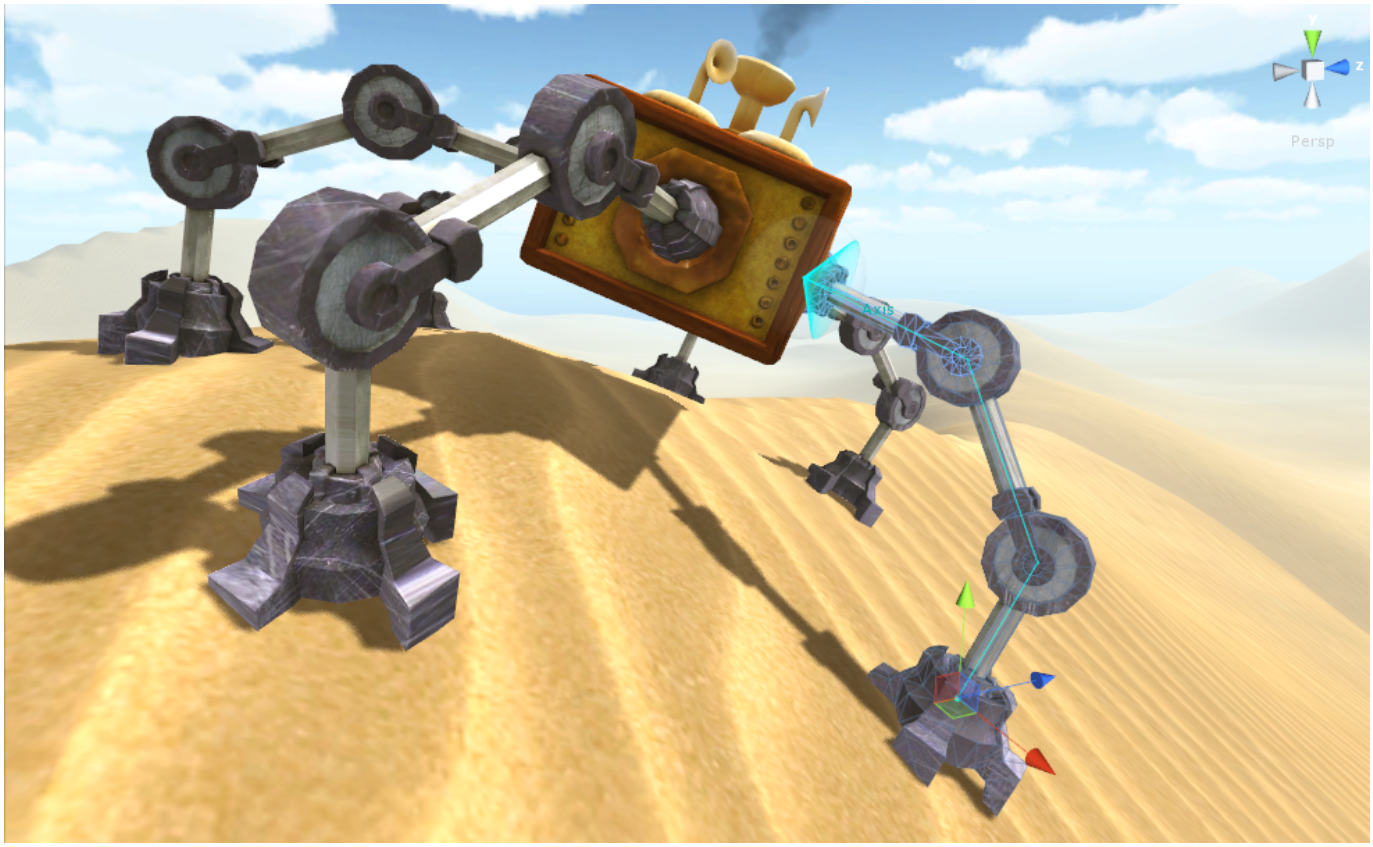
- Add the CCDIK component to the first GameObject in the chain
- Assign all the elements in the chain to "Bones" in the component
- Press Play, set weight to 1

Changing the target position:

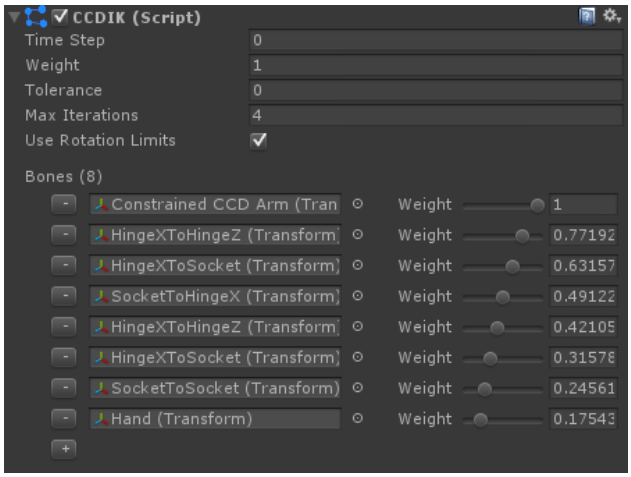
```
public CCDIK ccdIK;

void LateUpdate () {
    ccdIK.solver.IKPosition = something;
}
```

Adding CCDIK in runtime:

- Add the CCDIK component via script
- Call CCDIK.solver.SetChain()



CCD with rotation limits applied



The CCDIK component

FABRIK

Forward and Backward Reaching Inverse Kinematics solver based on the paper:
"FABRIK: A fast, iterative solver for the inverse kinematics problem."
Aristidou, A., Lasenby, J. Department of Engineering, University of Cambridge, Cambridge CB2 1PZ, UK.

FABRIK is a heuristic solver that can be used with any number of bone segments and rotation limits. It is a method based on forward and backward iterative movements by finding a joint's new position along a line to the next joint. FABRIK proposes to solve the IK problem in position space, instead of the orientation space, therefore it demonstrates less continuity under orientation constraints than CCD, although certain modifications have been made to the constraining method described in the original paper to improve solver stability. It generally takes less iterations to reach the target than CCD, but is slower per iteration especially with rotation limits applied.

FABRIK is extremely flexible, it even allows for direct manipulation of the bone segments in the scene view and the solver will readapt. Bone lengths can also be changed in runtime if updateBoneLengths parameter is set to true in the solver (false by default for performance reasons).

Monitoring and validating the IK chain each frame would be expensive on the performance, therefore changing the bone hierarchy in runtime has to be followed by calling SetChain (Transform[] hierarchy) on the solver. SetChain returns true if the hierarchy is valid.

Getting started:

- Add the FABRIK component to the first GameObject in the chain
- Assign all the elements in the chain to "Bones" in the component
- Press Play, set weight to 1

Changing the target position:

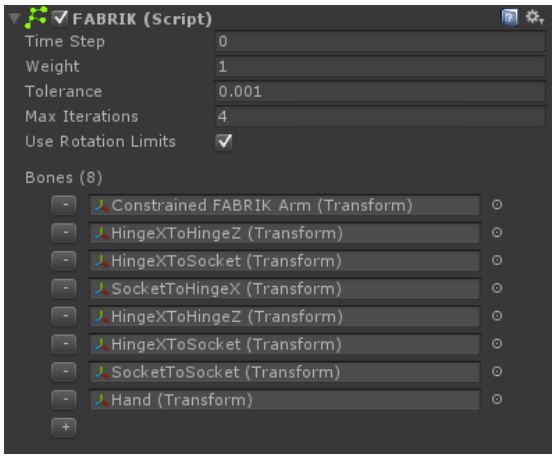
```
public FABRIK fabrik;
```

```
void LateUpdate () {
    fabrik.solver.IKPosition = something;
}
```

Adding FABRIK in runtime:

- Add the FABRIK component via script
- Call FABRIK.solver.SetChain()

FABRIK with rotation limits applied



The FABRIK component

FABRIK Root

Multi-effector FABRIK system.

FABRIKRoot is a component that connects FABRIK chains together to form extremely complicated IK systems with multiple branches, end-effectors and rotation limits.

Getting started:

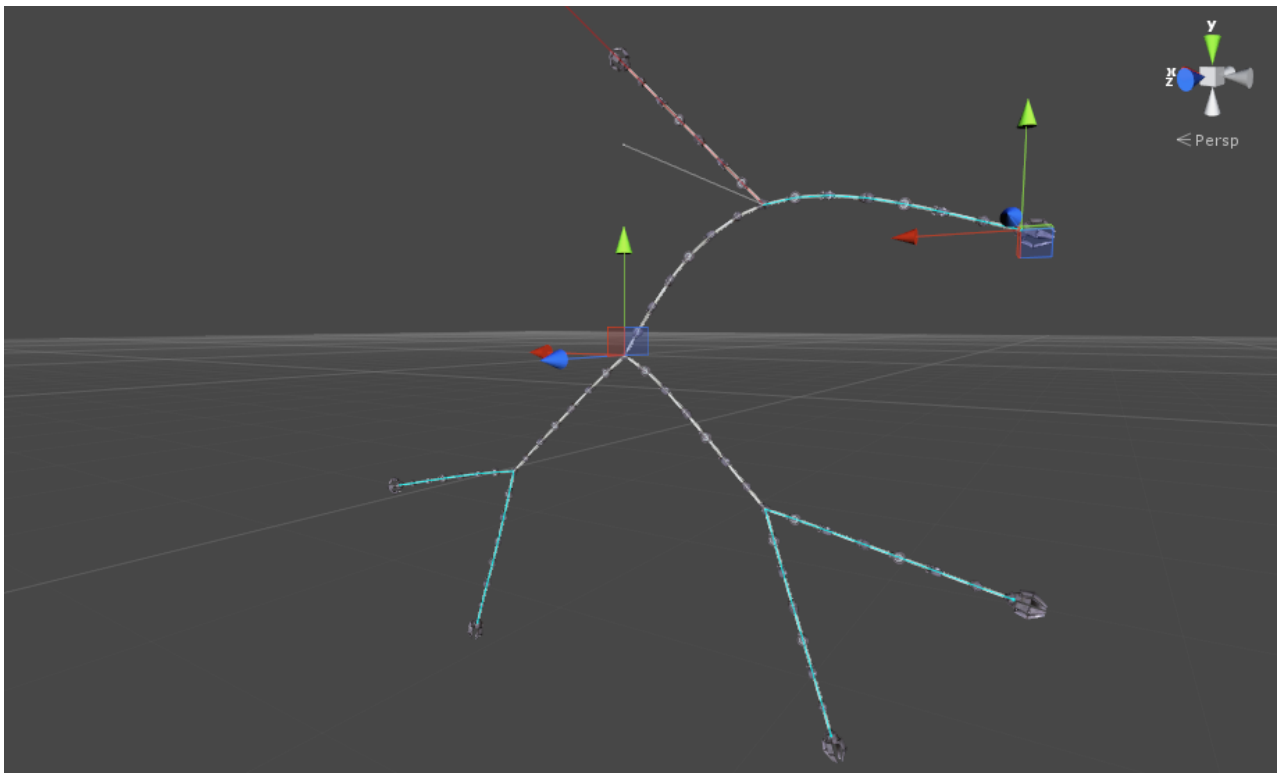
- Create multiple FABRIK chains, position them as you want them to be connected. The chains don't have to be parented to each other
- Make sure the first bone of a child chain is in the same position as the last bone of it's parent
- Create a new GameObject, add the FABRIKRoot component
- Add all the FABRIK chains to "Chains" in the FABRIKRoot component
- Press Play

Accessing the chains of FABRIKRoot:

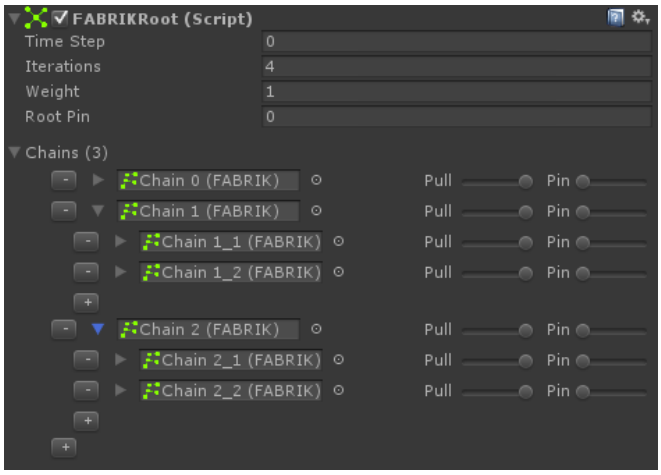
```
public FABRIKRoot fabrikRoot;

void LateUpdate () {
    Debug.Log(fabrikRoot.solver.chains[index].ik.name);
}
```

Limitations:

- Seperate FABRIK chains can not use the same bones, they must be fully independent
- The last bone of a FABRIK chain must be in the same position as it's child chain's first bone



FABRIK Root chain being pulled



The FABRIKRoot component

Full Body Biped IK

Final IK includes an extremely flexible and powerful high speed lightweight FBIK solver for biped characters.

FullBodyBipedIK maps any biped character to a low resolution multi-effector IK rig, solves it, and maps the result back to the character. This is done each frame in LateUpdate, after Mecanim/Legacy is done animating, so it is completely independent from the animating system.

Chains: Internally, each limb and the body are instances of the FBIKChain class. The root chain is the body, consisting of a single node, and the limbs are it's children. This setup forms the multi-effector IK tree around the root node.

Nodes: Nodes are members of the Chains. For instance, an Arm chain contains three nodes - upper arm, forearm and the hand. Each node maintains a reference to it's bone (node.transform). When the solver is processing or has finished, the solved position of the bone is stored in node.solverPosition.

Effectors: FullBodyBipedIK has three types of effectors - end-effectors (hands and feet), mid-body effectors (shoulders and thighs) and multi-effectors (the body). End-effectors can be rotated while changing the rotation of mid-body and multi-effectors has no effect. Changing end-effector rotation also changes the bending direction of the limb (unless you are using bend goals to override it). The body effector is a multi-effector, meaning it also drags along both thigh effectors (to simplify positioning of the body). Effectors also have the positionOffset property that can be used to very easily manupulate with the underlaying animation. Effectors will reset their positionOffset to Vector3.zero after each solver update.

Pulling and Reaching: Each chain has the "pull" property. When all chains have pull equal to 1, pull weight is distributed equally between the limbs. That means reaching all effectors is not quaranteed if they are very far from each other. The result can be adjusted or improved by changing the "reach" parameter of the chain, increasing the solver iteration count or updating the solver more than once per frame. However, when for instance the left arm chain has pull weight equal to 1 and all others have 0, you can pull the character from it's left hand to Infinity without losing contact.

Mapping: IKSolverFullBodyBiped solves a very low resolution high speed armature. Your character probably has a lot more bones in it's spine though, it might have twist bones in the arms and shoulder or hip bones and so on. Therefore, the solver needs to map the high resolution skeleton to the low resolution solver skeleton before solving and vice versa after the solver has finished. There are 3 types of mappers - IKMappingSpine for mapping the pelvis and the spine, IKMappingLimb for the limbs (including the clavicle) and IKMappingBone for the head. You can access them through IKSolverFullBody.spineMapping, IKSolverFullBody.limbMappings and IKSolverFullBody.boneMappings

Limitations:

- FullBodyBipedIK does not have an effector for the head. That is because the head is just a bone that you can simply rotate however you please after Full Body IK is finished, and there are very few cases, where you would actually need to pull a character from the head. Even then, it could be simulated just as well by pulling the shoulder effectors instead. This is an optimisation that grants us more speed and stability.
- FullBodyBipedIK does not have effectors for the fingers and toes. Solving fingers with IK would be an overkill in most cases as there are only so few poses for the hands in a game. Using 10 4-segment constrained CCD or FABRIK chains to position the fingers however is probably something you don't want to waste your precious milliseconds on. See the Driving Rig demo to get an idea how to very quickly (and entirely in Unity) pose the fingers to an object.
- FullBodyBipedIK samples the initial pose of your character (in Start() and each time you re-initiate the solver) to find out which way the limbs should be bent. Hence the limitation - the limbs of the character at that moment should be bent in their natural directions. Some characters however are in geometrically perfect T-Pose, meaning their limbs are completely straight. Some characters even have their limbs bent slightly in the inverse direction (some Mixamo rigs for example). FullBodyBipedIK will alarm you should this problem occur. All you will have to do, is rotate the forearm or calf bones in the Scene view slightly in the direction they should be bent. Since those rotations will be overwritten in play mode by animation anyway, you should not be afraid of messing up your character.
- FullBodyBipedIK does not have elbow/knee effectors. That might change in the future should there be a practical demand for them. Elbow and knee positions can still be modified though as bend goals are supported.
- Optimize Game Objects should be disabled or at least all the bones needed by the solver (FullBodyBipedIK.references) exposed.
- Additional bones in the limbs are supported as long as their animation is twisting only. If the additional bones have swing animation, like for example wing bones, FBBIK will not solve the limb correctly.
- FullBodyBipedIK works with characters that have animatePhysics enabled. There is a bug in Mecanim though, that does not allow for changing animatePhysics in runtime, therefore FullBodyBipedIK will update according to the initial animatePhysics state.
- FullBodyBipedIK does not rotate the shoulder bone when the character is pulled by the hand. It will maintain the shoulder bone rotation relative to the chest as it is in the animation. In most cases, it is not a problem, but sometimes, especially when reaching for something above the head, having the shoulder bone rotate along would make it more realistic. In this case you should either have an underlaying reach up animation that rotates the shoulder bone or it can also be rotated via script before the IK solver reads the character's pose. There is also a workaround script included in the demos, called ShoulderRotator.
- When you move a limb end-effector and the effector rotation weight is 0, FBBIK will try to maintain the bending direction of the limb as it is animated. When the limb rotates close to 180 degrees from it's animated direction, you will start experiencing rolling of the limb, meaning, the solver has no way to know at this point of singularity, which way to rotate the limb. Therefore if you for example have a walking animation, where the hands are down and you want to use IK to grab something from directly above the head, you will have to take the inconvenience to also animate the effector rotation or use a bend goal, to make sure the arm does not roll backwards when close to 180 degrees of angular offset. This is not a bug, it is a logical inevitability if we want to maintain the animated bending direction by default.
- FullBodyBipedIK considers all elbows and knees as 3 DOF joints with swing rotation constrained to the range of a hemisphere (Since 0.22, used to be 1 DOF). That allows for full accuracy mapping of all biped rigs, the only known limitation is that the limbs can't be inverted (broken from the knee/elbow).

Getting started:

- Add the FullBodyBipedIK component to the root of your character (the same GameObject that has the Animator/Animation component)
- Make sure the auto-detected biped references are correct
- Make sure the Root Node was correctly detected. It should be one of the bones in the lower spine.
- Take a look at the character in the scene view, make sure you see the FullBodyBipedIK armature on top the character.
- Press Play, weigh in the solvers

Accessing the Effectors:

```
public FullBodyBipedIK ik;

void LateUpdate () {
    ik.solver.leftHandEffector.position = something; // Set the left hand effector position to a point in world space. This has no
       effect if the effector's positionWeight is 0.
    ik.solver.leftHandEffector.rotation = something; // Set the left hand effector rotation to a point in world space. This has no
       effect if the effector's rotationWeight is 0.
    ik.solver.leftHandEffector.positionWeight = 1f; // Weighing in the effector position, the left hand will be pinned to
       ik.solver.leftHandEffector.position.

    // Weighing in the effector rotation, the left hand and the arm will be pinned to ik.solver.leftHandEffector.rotation.
    // Note that if you only wanted to rotate the hand, but not change the arm bending,
    // it is better to just rotate the hand bone after FBBIK has finished updating (use the OnPostUpdate delegate).
    ik.solver.leftHandEffector.rotationWeight = 1f;

    // Offsets the hand from it's animated position. If effector positionWeight is 1, this has no effect.
    // Note that the effectors will reset their positionOffset to Vector3.zero after each update, so you can (and should) use them
       additively.
    //This enables you to easily edit the value by more than one script.
    ik.solver.leftHandEffector.positionOffset += something;

    //The effector mode is for changing the way the limb behaves when not weighed in.
    //Free means the node is completely at the mercy of the solver.
    //(If you have problems with smoothness, try changing the effector mode of the hands to MaintainAnimatedPosition or
       MaintainRelativePosition

    //MaintainAnimatedPosition resets the node to the bone's animated position in each internal solver iteration.
    //This is most useful for the feet, because normally you need them where they are animated.

    //MaintainRelativePositionWeight maintains the limb's position relative to the chest for the arms and hips for the legs.
    // So if you pull the character from the left hand, the right arm will rotate along with the chest.
    //Normally you would not want to use this behaviour for the legs.
    ik.solver.leftHandEffector.maintainRelativePositionWeight = 1f;

    // The body effector is a multi-effector, meaning it also manipulates with other nodes in the solver, namely the left thigh and the
       right thigh
    // so you could move the body effector around and the thigh bones with it. If we set effectChildNodes to false, the thigh nodes
       will not be changed by the body effector.
    ik.solver.body.effectChildNodes = false;

    // Other effectors: rightHandEffector, leftFootEffector, rightFootEffector, leftShoulderEffector, rightShoulderEffector,
       leftThighEffector, rightThighEffector, bodyEffector

    // You can also find an effector by:
    ik.solver.GetEffector(FullBodyBipedEffector effectorType);
    ik.solver.GetEffector(FullBodyBipedChain chainType);
    ik.solver.GetEndEffector(FullBodyBipedChain chainType); // Returns only hand or feet effectors
}
```

Accessing the Chains:

```
public FullBodyBipedIK ik;

void LateUpdate () {
    ik.solver.leftArmChain.pull = 1f; // Changing the Pull value of the left arm
    ik.solver.leftArmChain.reach = 0f; // Changing the Reach value of the left arm

    // Other chains: rightArmChain, leftLegChain, rightLegChain, chain (the root chain)

    // You can also find a chain by:
    ik.solver.GetChain(FullBodyBipedChain chainType);
    ik.solver.GetChain(FullBodyBipedEffector effectorType);
}
```

Accessing the Mapping:

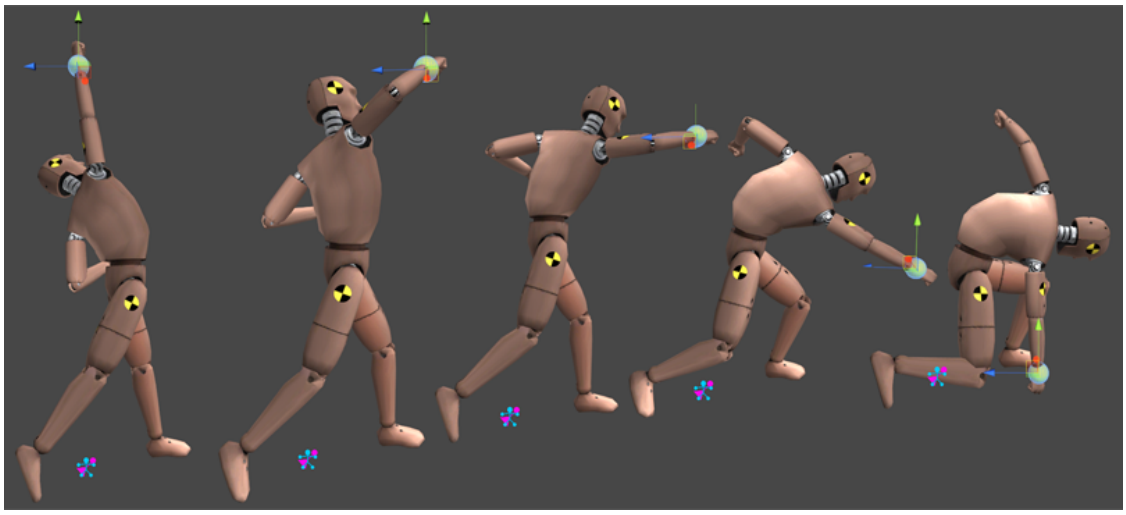
```
public FullBodyBipedIK ik;

void LateUpdate () {
    ik.solver.spineMapping.iterations = 2; // Changing the Spine Mapping Iterations
    ik.solver.leftArmMapping.maintainRotationWeight = 1f; // Make the left hand maintain it's rotation as animated.
    ik.solver.headMapping.maintainRotationWeight = 1f; // Make the head maintain it's rotation as animated.
}
```

Adding FullBodyBipedIK in runtime (UMA):

```
using RootMotion; // Need to include the RootMotion namespace as well because of the BipedReferences

FullBodyBipedIK ik;

// Call this method whenever you need in runtime.
// Please note that FBBIK will sample the pose of the character at initiation so at the time of calling this method,
// the limbs of the character should be bent in their natural directions.
void AddFBBIK (GameObject go, BipedReferences references = null) {

    if (references == null) { // Auto-detect the biped definition if we don't have it yet
        BipedReferences.AutoDetectReferences(ref references, go.transform, BipedReferences.AutoDetectParams.Default);
    }

    ik = go.AddComponent<FullBodyBipedIK>(); // Adding the component

    // Set the FBBIK to the references. You can leave the second parameter (root node) to null if you trust FBBIK to automatically set
       it to one of the bones in the spine.
    ik.SetReferences(references, null);

    // Using pre-defined limb orientations to safeguard from possible pose sampling problems (since 0.22)
    ik.solver.SetLimbOrientations(BipedLimbOrientations.UMA); // The limb orientations definition for UMA skeletons
    // or...
    ik.solver.SetLimbOrientations(BipedLimbOrientations.MaxBiped); // The limb orientations definition for 3ds Max Biped skeletons
    // or..
    ik.solver.SetLimbOrientations(yourCustomBipedLimbOrientations); // Your custom limb orientations definition

    // To know how to fill in the custom limb orientations definition, you should imagine your character standing in I-pose (not T-
       pose) with legs together and hands on the sides...
    // The Upper Bone Forward Axis is the local axis of the thigh/upper arm bone that is facing towards character forward.
    // Lower Bone Forward Axis is the local axis of the calf/forearm bone that is facing towards character forward.
    // Last Bone Left Axis is the local axis of the foot/hand that is facing towards character left.
}
```

Optimizing FullBodyBipedIK:

- You can use renderer.isVisible to weigh out the solver when the character is not visible.
- Most of the time you don't need so many solver iterations and spine mapping iterations. Note that with just 1 iteration character shoulders and thighs might get dislocated when pulled from both hands/feet
- Keep the "Reach" values at 0 if you don't need them. By default they are 0.05f to improve accuracy.
- Keep the Spine Twist Weight at 0 if you don't see the need for it.
- Also setting the "Spine Stiffness", "Pull Body Vertical" and/or "Pull Body Horizontal" to 0 will help the performance.



Retargeting a single punching animation with FullBodyBipedIK

The FullBodyBipedIK component

Limb

LimbIK extends TrigonometricIK to specialize on the 3-segmented hand and leg character limb types. LimbIK comes with multiple Bend Modifiers:

- Animation: tries to maintain bend direction as it is in the animation
- Target: rotates the bend direction with the target IKRotation
- Parent: rotates the bend direction along with the parent Transform (pelvis or clavicle)
- Arm: keeps the arm bent in a biometrically natural and relaxed way (also most expensive of the above).

If none of the automatic bend modifiers fit your needs, you can always create bend goal objects. It is as easy as:

```
using RootMotion.FinalIK;

public LimbIK limbIK;

void LateUpdate () {
    limbIK.solver.SetBendGoalPosition(transform.position);
}
```

This will make the limb bend towards the direction from the first bone to the position of the goal.

The IKSolverLimb.maintainRotationWeight property allows to maintain the world space rotation of the last bone fixed as it was before solving the limb.
This is most useful when we need to reposition a foot, but maintain it's rotation as it was animated to ensure proper alignment with the ground surface.

Getting started:

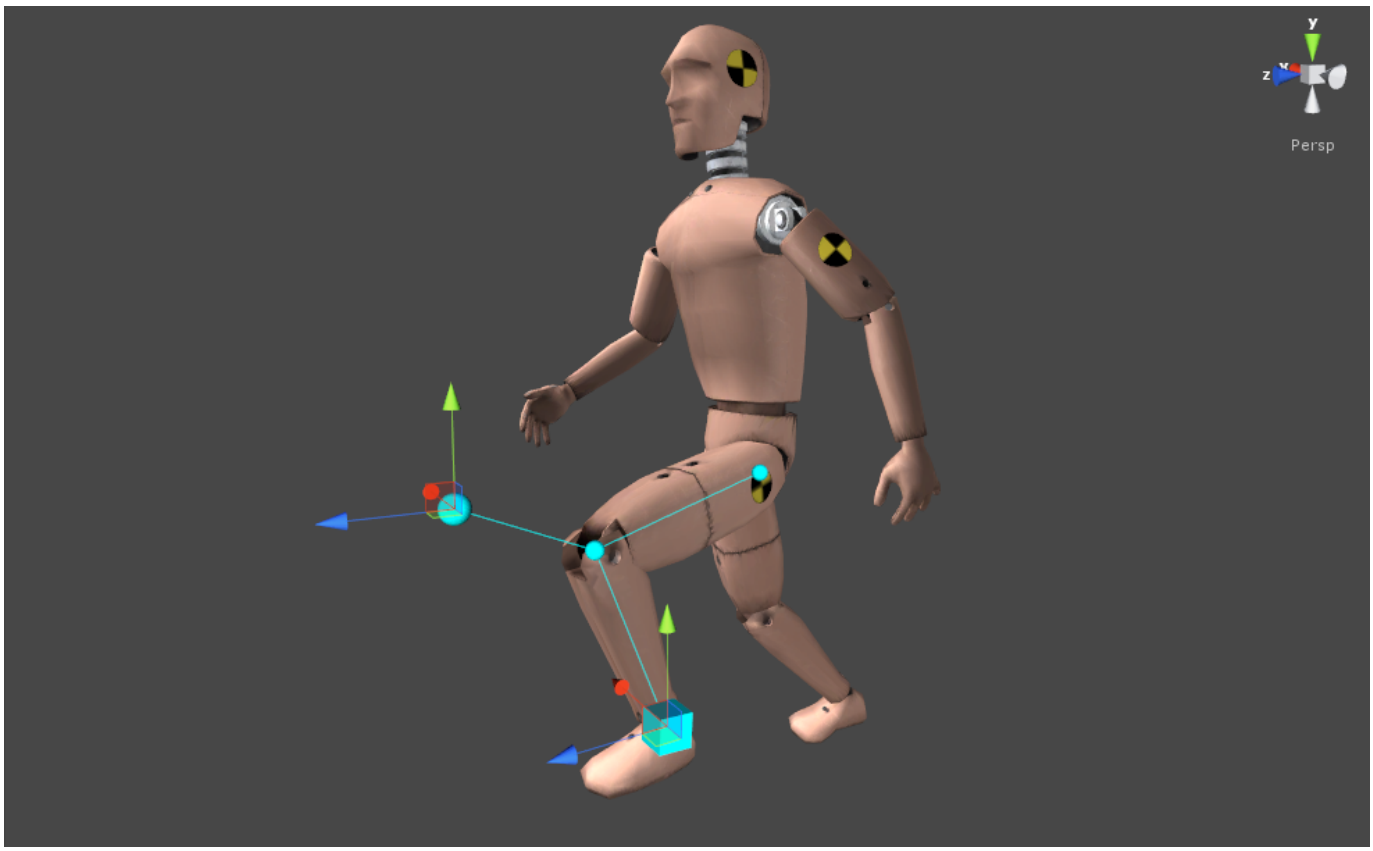
- Add the LimbIK component to the root of your character (the character should be facing it's forward direction)
- Assign the limb bones to bone1, bone2 and bone3 in the LimbIK component
- Press Play

Getting started with scripting:

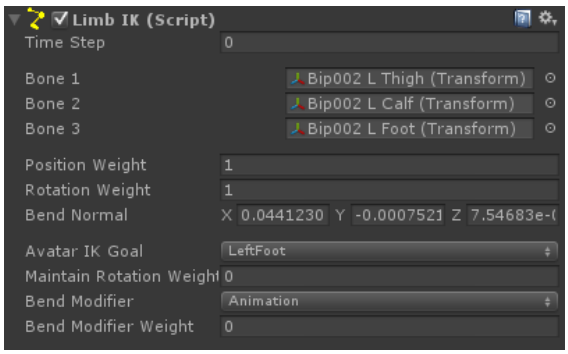
```
public LimbIK limbIK;

void LateUpdate () {
    // Changing the target position, rotation and weights
    limbIK.solver.IKPosition = something;
    limbIK.solver.IKRotation = something;
    limbIK.solver.IKPositionWeight = something;
    limbIK.solver.IKRotationWeight = something;

    // Changing the automatic bend modifier
    limbIK.solver.bendModifier = IKSolverLimb.BendModifier.Animation; // Will maintain the bending direction as it is animated.
    limbIK.solver.bendModifier = IKSolverLimb.BendModifier.Target; // Will bend the limb with the target rotation
    limbIK.solver.bendModifier = IKSolverLimb.BendModifier.Parent; // Will bend the limb with the parent bone (pelvis or shoulder)

    // Will try to maintain the bend direction in the most biometrically relaxed way for the arms.
    // Will not work for the legs.
    limbIK.solver.bendModifier = IKSolverLimb.BendModifier.Arm;
}
```

Adding LimbIK in runtime:

- Add the LimbIK component via script
- Call LimbIK.solver.SetChain()



LimbIK with bend goal



The LimbIK component

LookAt

LookAt IK can be used on any character or other hierarchy of bones to rotate a set of bones to face a target.

Getting started:

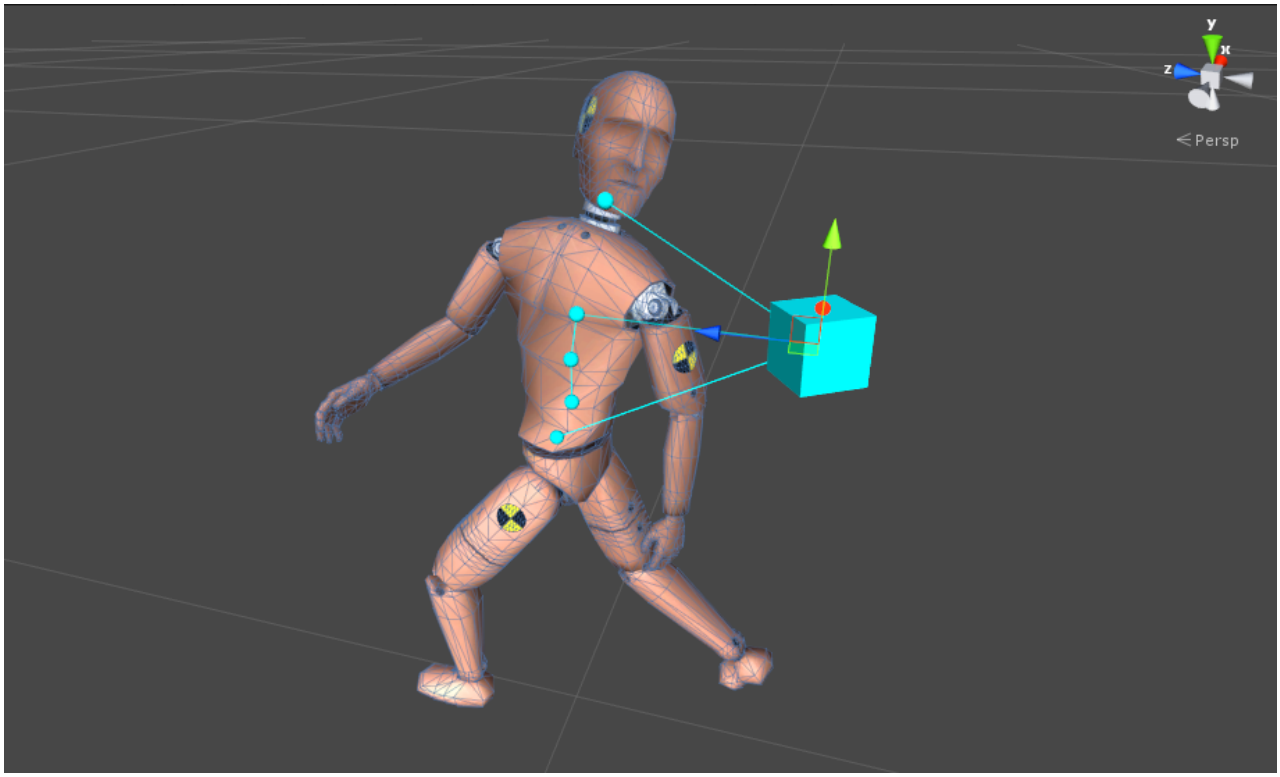
- Add the LookAtIK component to the root GameObject. That GameObject's forward axis will be the forward direction.
- Assing Spine, head and eye bones to the component.
- Press Play

Getting started with scripting:

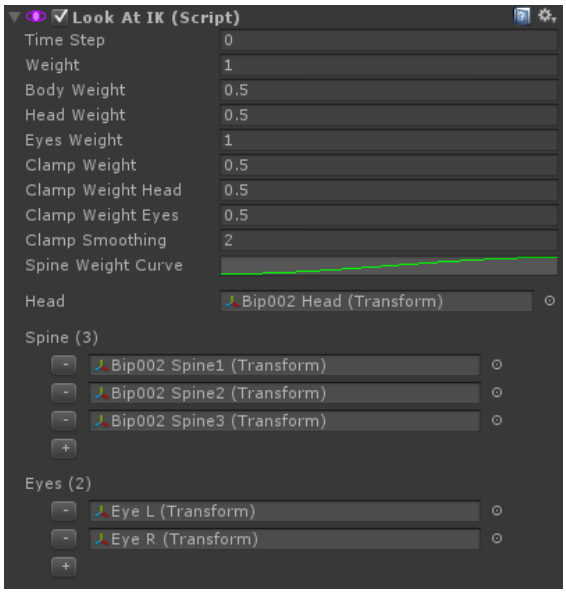
```
public LookAtIK lookAt;

void LateUpdate () {
    lookAt.solver.IKPositionWeight = 1f; // The master weight

    lookAt.solver.IKPosition = something; // Changing the look at target

    // Changing the weights of individual body parts
    lookAt.solver.bodyWeight = 1f;
    lookAt.solver.headWeight = 1f;
    lookAt.solver.eyesWeight = 1f;

    // Changing the clamp weight of individual body parts
    lookAt.solver.clampWeight = 1f;
    lookAt.solver.clampWeightHead = 1f;
    lookAt.solver.clampWeightEyes = 1f;
}
```

Adding LookAtIK in runtime:

- Add the LookAtIK component via script
- Call LookAtIK.solver.SetChain()



LookAtIK in action



The LookAtIK component

Trigonometric

Trigonometric IK is the most basic IK solver that is based on the Law of Cosines and solves a 3-segmented bone hierarchy.

Getting started:

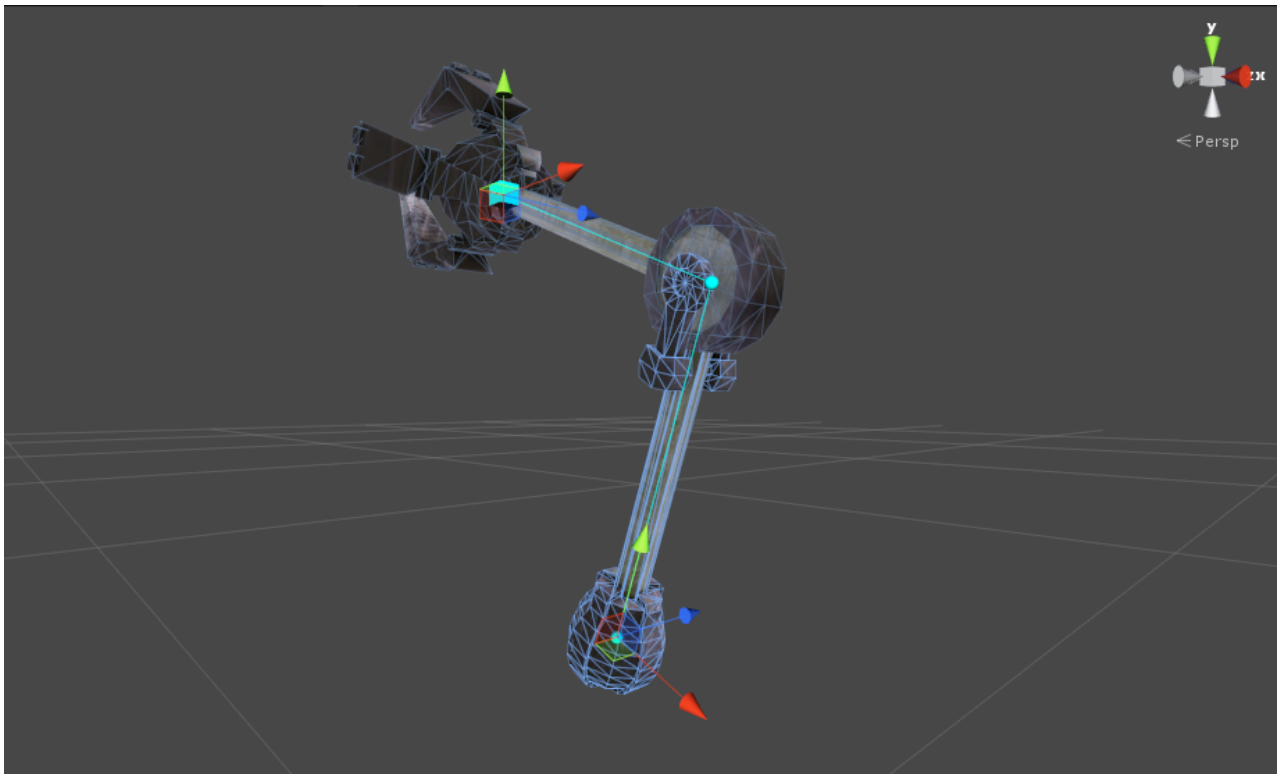
- Add the TrigonometricIK component to the first bone.
- Assign bone1, bone2 and bone3 in the TrigonometricIK component
- Press Play

Getting started with scripting:

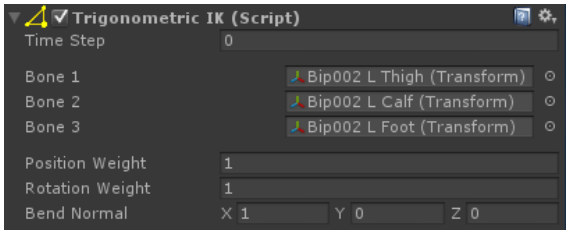
```
public TrigonometricIK trig;

void LateUpdate () {
    // Changing the target position, rotation and weights
    trig.solver.IKPosition = something;
    trig.solver.IKRotation = something;
    trig.solver.IKPositionWeight = something;
    trig.solver.IKRotationWeight = something;

    trig.solver.SetBendGoalPosition(Vector goalPosition); // Sets the bend goal to a point in world space
}
```

Adding TrigonometricIK in runtime:

- Add the TrigonometricIK component via script
- Call TrigonometricIK.solver.SetChain()



Solving a 3-segment chain with TrigonometricIK



The TrigonometricIK component

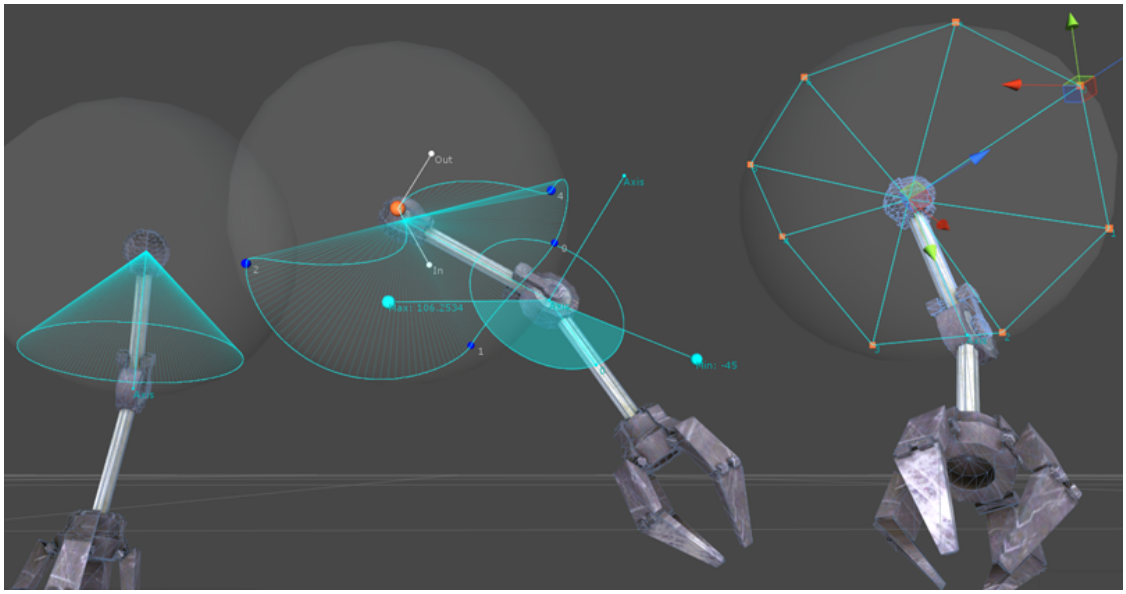
Rotation Limits

All rotation limits and other Final IK components are Quaternion and Axis-Angle based to ensure consistency, continuity and to minimize singularity issues. Final IK does not contain a single Euler operation.

All rotation limits are based on local rotations and use the initial local rotation as reference just like Physics joints. This makes them axis-independent and intuitive to set up.

All rotation limits have undoable Scene view editors.

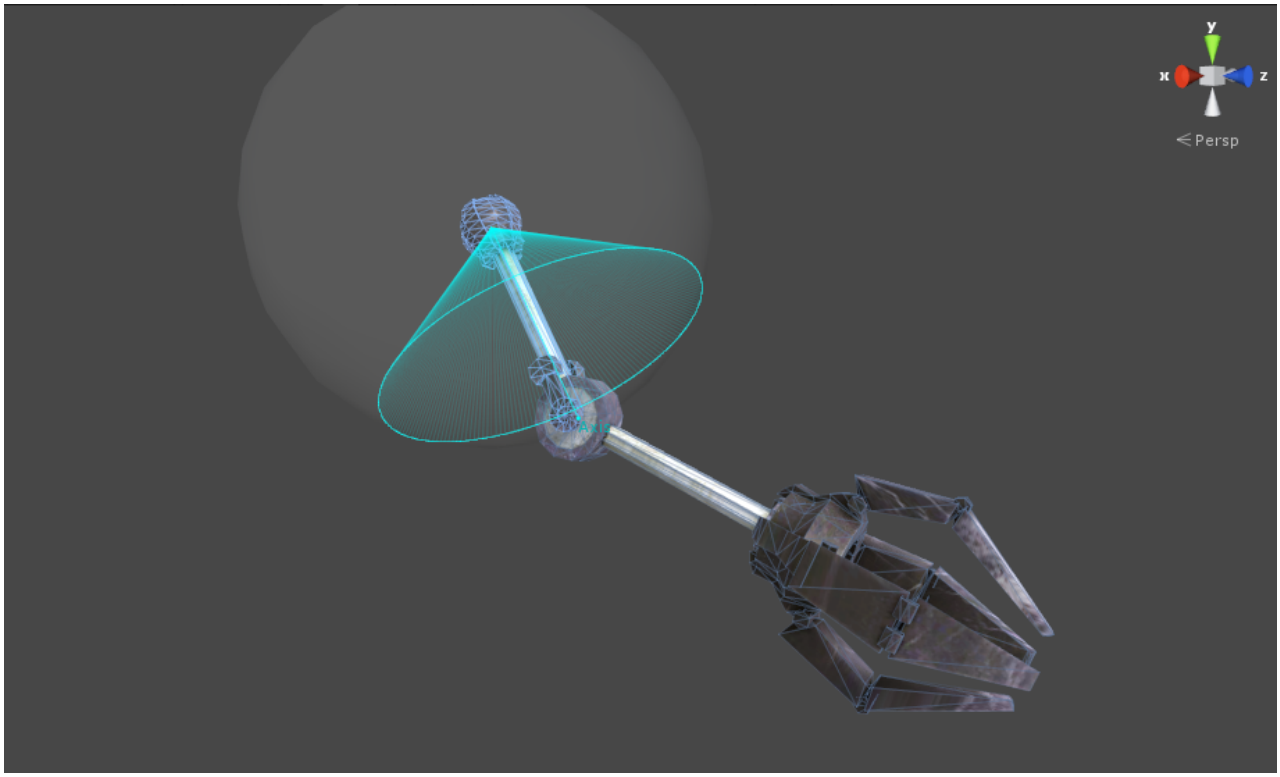
All rotation limits work with IK solvers that support rotation limits.



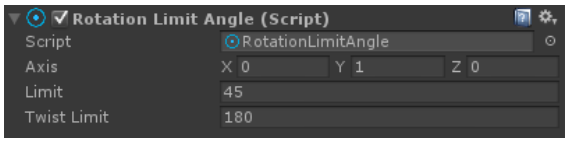
Rotation Limits

Angle

Simple angular swing and twist limit.



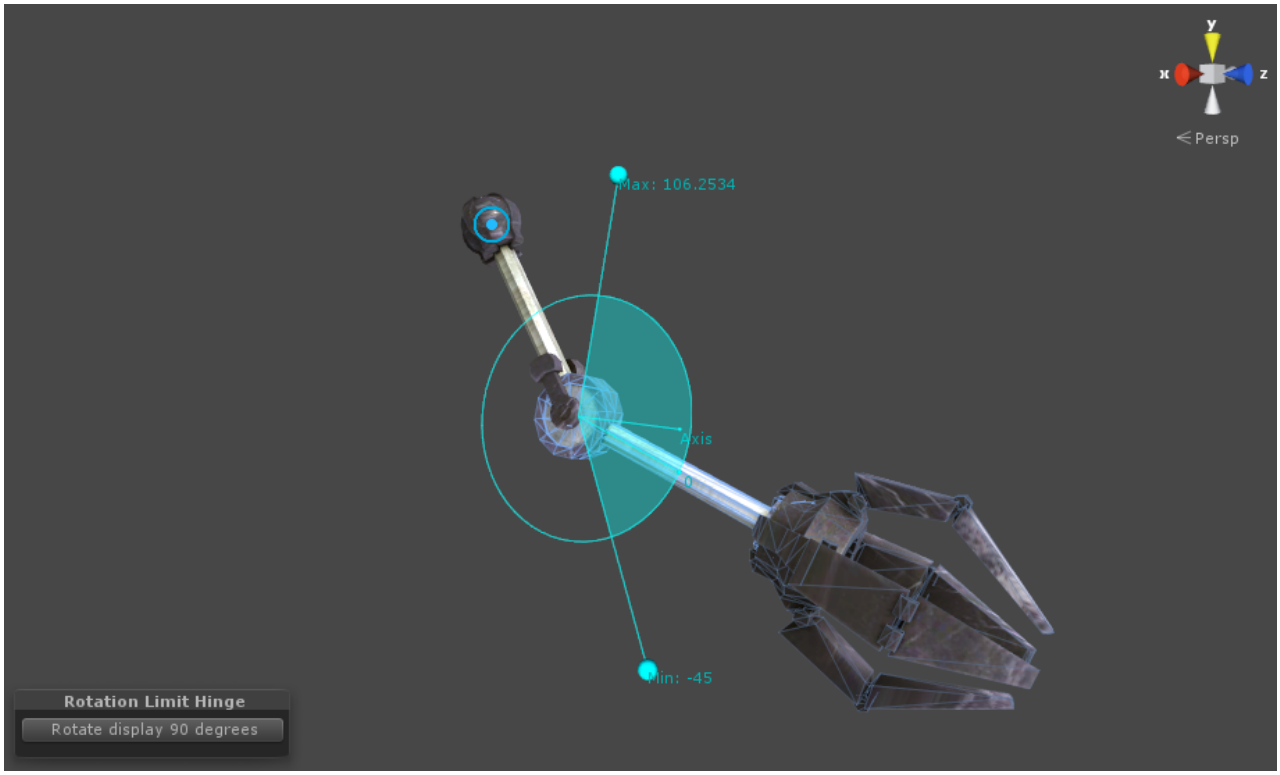
The anglular rotation limit



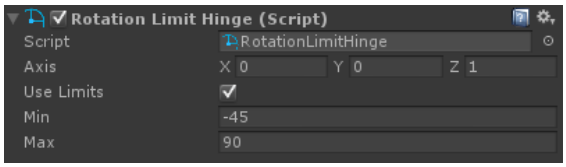
The RotationLimitAngle component

Hinge

The hinge rotation limit limits the rotation to a single degree of freedom around an axis. This rotation limit is additive which means the hinge limits can exceed 360 degrees either way.



Adjusting hinge limits in the scene view



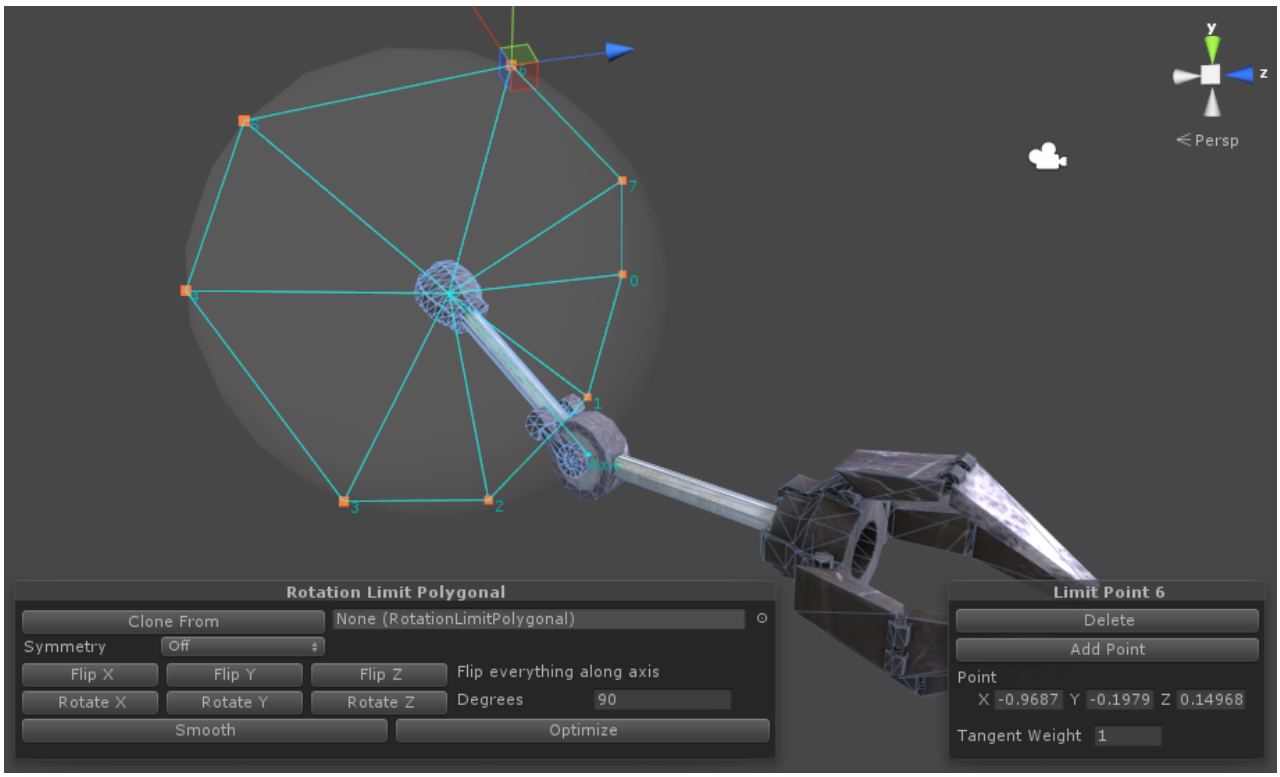
The RotationLimitHinge component

Polygonal

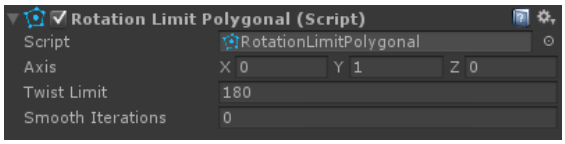
Using a spherical polygon to limit the range of rotation on universal and ball-and-socket joints. A reach cone is specified as a spherical polygon on the surface of a a reach sphere that defines all positions the longitudinal segment axis beyond the joint can take.
The twist limit parameter specifies the maximum twist around the main axis.

This class is based on the paper:
"Fast and Easy Reach-Cone Joint Limits"
Jane Wilhelms and Allen Van Gelder. Computer Science Dept., University of California, Santa Cruz, CA 95064. August 2, 2001

The polygonal rotation limit is provided with handy scene view tools for quick editing, cloning and modifying of the reach cone points.



Defining reach cone points on the polygonal rotation limit



RotationLimitPolygonal component

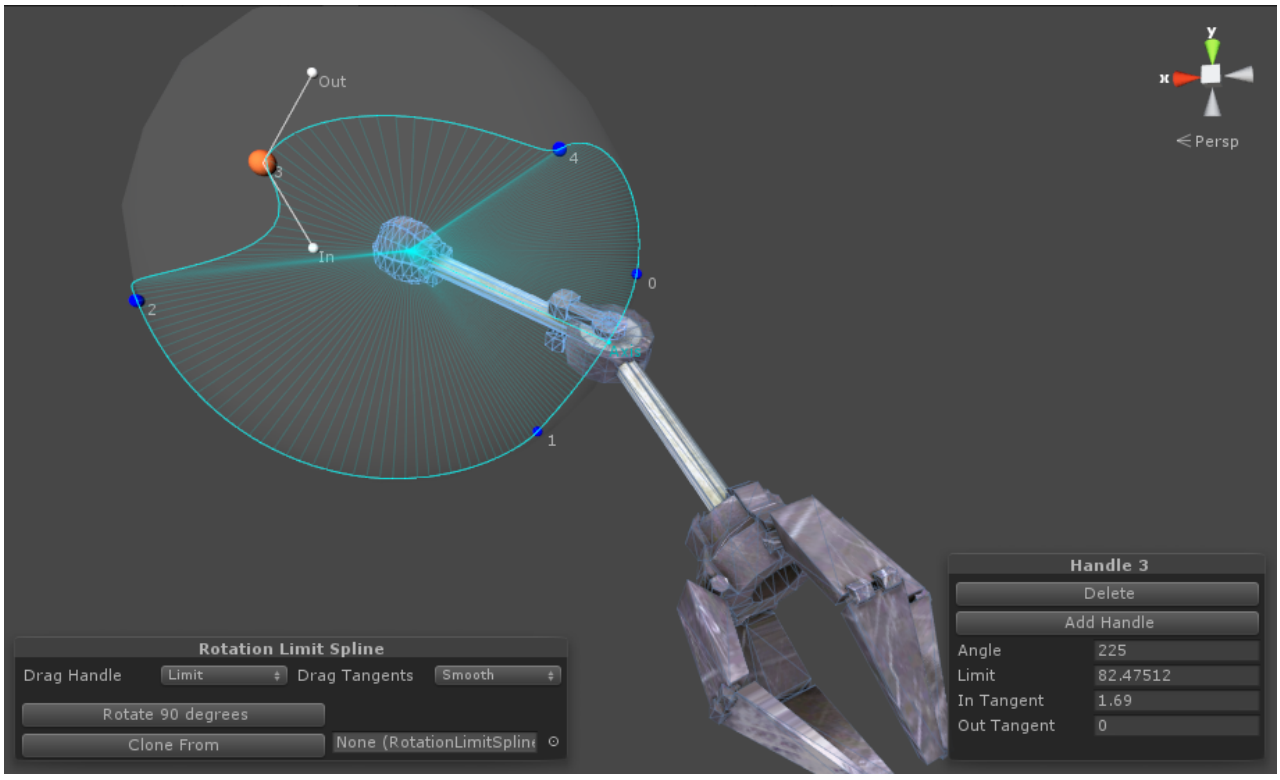
Spline

Using a spline to limit the range of rotation on universal and ball-and-socket joints.

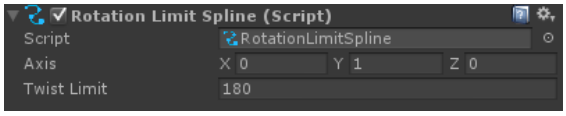
Reachable area is defined by an AnimationCurve orthogonally mapped onto a sphere, which provides a very smooth and fast result.

The twist limit parameter specifies the maximum twist around the main axis.

The spline rotation limit is provided with handy scene view tools for quick editing, cloning and modifying of the spline handles.



Adjusting spline handles on on the spline rotation limit



The RotationLimitSpline component

Extending Final IK

The IK solvers and rotation limits of FinalIK were built from the ground up with extendability in mind.

Some of the components of FinalIK, such as BipedIK, are essentially little more than just collections of IK solvers.

Writing Custom IK Components

Before you can exploit the full power of FinalIK, it is important to know a few things about it's architecture.

The difference between IK components and IK solvers:

By architecture, IK solver is a class that actually contains the inverse kinematics functionality, while the function of an IK component is only to harbor, initiate and update it's solver and provide helpful scene view handles as well as custom inspectors.

Therefore, IK solvers are fully independent of their components and can even be used without them through direct reference:

```
using RootMotion.FinalIK;

public IKSolverCCD spine = new IKSolverCCD();
public IKSolverLimb limb = new IKSolverLimb();

void Start() {
    // The root transform reference is used in the initiation of IK solvers for multiple reasons depending on the solver.
    // heuristic solvers IKSolverCCD, IKSolverFABRIK and IKSolverAim only need it as context for logging warnings,
    // character solvers IKSolverLimb, IKSolverLookAt, BipedIK and IKSolverFullBodyBiped use it to define their orientation relative to the character,
    // IKSolverFABRIKRoot uses it as the root of all of it's FABRIK chains.
    spine.Initiate(transform);
    limb.Initiate(transform);
}

void LateUpdate() {
    // Updating the IK solvers in a specific order.
    // In the case of multiple IK solvers handling a bone hierarchy, it is usually wise to solve the parents first.
    spine.Update();
    limb.Update();
}
```

You now have essentially a custom IK component.

This can be helpful if you needed to keep all the functionality of your IK system in a single component, like BipedIK, so you would not have to manage many different IK components in your scene.

Writing Custom Rotation Limits

All rotation limits in Final IK extend from the abstract RotationLimit class. To compose your own, you would as well need to extend from this base class and override the abstract method

```
protected abstract Quaternion LimitRotation(Quaternion rotation);
```

In this method you will have to apply the constraint to and return the input Quaternion.

It is important to note that the input Quaternion is already converted to the default local rotation space of the gameobject, meaning if you return Quaternion.identity, the gameobject will always remain fixed to it's initial local rotation.

The following code could be a template for a custom rotation limit:

```
using RootMotion.FinalIK;

// Declaring the class and extending from RotationLimit.cs
public class RotationLimitCustom: RotationLimit {

    // Limits the rotation in the local space of this instance's Transform.
    protected override Quaternion LimitRotation(Quaternion rotation) {
        return MyLimitFunction(rotation);
    }

}
```

The new rotation limit gets recognized and applied automatically by all constrainable IK solvers.

Combining IK Components

When creating more complex IK systems, you will probably need full control over the updating order of your solvers. To do that, you can just disable their components and manage their solvers from an external script.

All IK components extend from the abstract IK class and all IK solvers extend from the abstract IKSolver class. This enables you to easily handle or replace the solvers even without needing to know the specific type of the solver.

Controlling the updating order of multiple IK components:

```
using RootMotion.FinalIK;

// Array of IK components that you can assign from the inspector.
// IK is abstract, so it does not matter which specific IK component types are used.
public IK[] components;

void Start() {
    // Disable all the IK components so they won't update their solvers. Use Disable() instead of enabled = false, the latter does not guarantee solver initiation.
    foreach (IK component in components) component.Disable();
}

void LateUpdate() {
    // Updating the IK solvers in a specific order.
    foreach (IK component in components) component.GetIKSolver().Update();
}
```