



IK Plus

v0.5.1

Developed by **Hristo Mihailov Ivanov**

(ASP: **Kris Development**)

Support e-mail:

krisdevmail@gmail.com

Introduction

IK Plus is an easy to use, yet advanced modular IK system that allows you to create realistic character movements with very little effort. It enables you to make your characters interact with the environment in more believable ways.

IK Plus has been developed with the end user in mind, featuring organized and readable interface, convenient script access functions and **optional** modules which you can add depending on the purpose of your IK.

Technical Note: The tool relies on the external library SETUtil. Also for scripting purposes you will have to include the IKPn and SETUtil namespace in your scripts.

<https://gitlab.com/KrisDevelopment/SETUtil>

Menu

Introduction.....	2
How to setup IKP.....	4
How to setup a module.....	4
The Generic Limb Module.....	6
The Head Module.....	8
The Upper Body Module.....	10
The Lower Body Module.....	13
The Weapon Module.....	15
The Weapon Collision Module.....	17
The Editor Simulation Module.....	18
IKP_Weapon script and weapon setup.....	18
Setting Targets & Properties through code.....	19
Module Properties Table.....	22
The IKP Blender.....	23
Using the Blend Machine Editor.....	24
Creating blend animations.....	26
IKP Types Library:.....	28
Standard enums:.....	28
Upper Body Module enums:.....	28
Weapon Module enums:.....	28
Important Custom Classes:.....	29
Dictionary.....	30

How to setup IKP

Select the root of the character armature (for example the player object), *which will act as a reference to the forward-facing vector*, and **add the IKP.cs script** either from the “Add Component” menu or by simply dragging it from the project panel to the inspector.

At first the IKP component is empty, it holds no modules. To add modules to it, click on the “**Add/Edit Module**” button, which will open a window with all available modules. Select the ones you need from the list. If you want to remove a module, click the “X” button next to the module name from that same panel.

After selecting the desired modules, **you will need to assign bone transform references** for each one.

How to setup a module

Almost all modules follow one general setup process, but before we begin let's define two very important terms:

“**Module panel**” is the graphic interface inside the IKP component. From it you can control all the important properties and settings of a module or enable and disable it.

“**Module component**” is the component that has been added in the inspector under the IKP. It is the actual “physical” object that executes the discrete IK logic.

*For the more complex modules, inside the module panel there are usually **three** categories – setup, settings and properties.

In **Setup** you will link bones and important references, or toggle major functionality that can't be altered at runtime.

In **Settings** will toggle features (such as feet raycasting etc.) and tweak minor behavior (forced positioning, etc.). Modifying anything in that panel should be compatible with runtime code and should not break or halt the IK. On various rare occasions you may need to toggle settings to better align with your animation context. Modifying these settings is not intended to be done frequently or as part of the standard use case, so programming access is through the Component: `GetComponent<SomeModule>().somePublicProperty = someValue`.

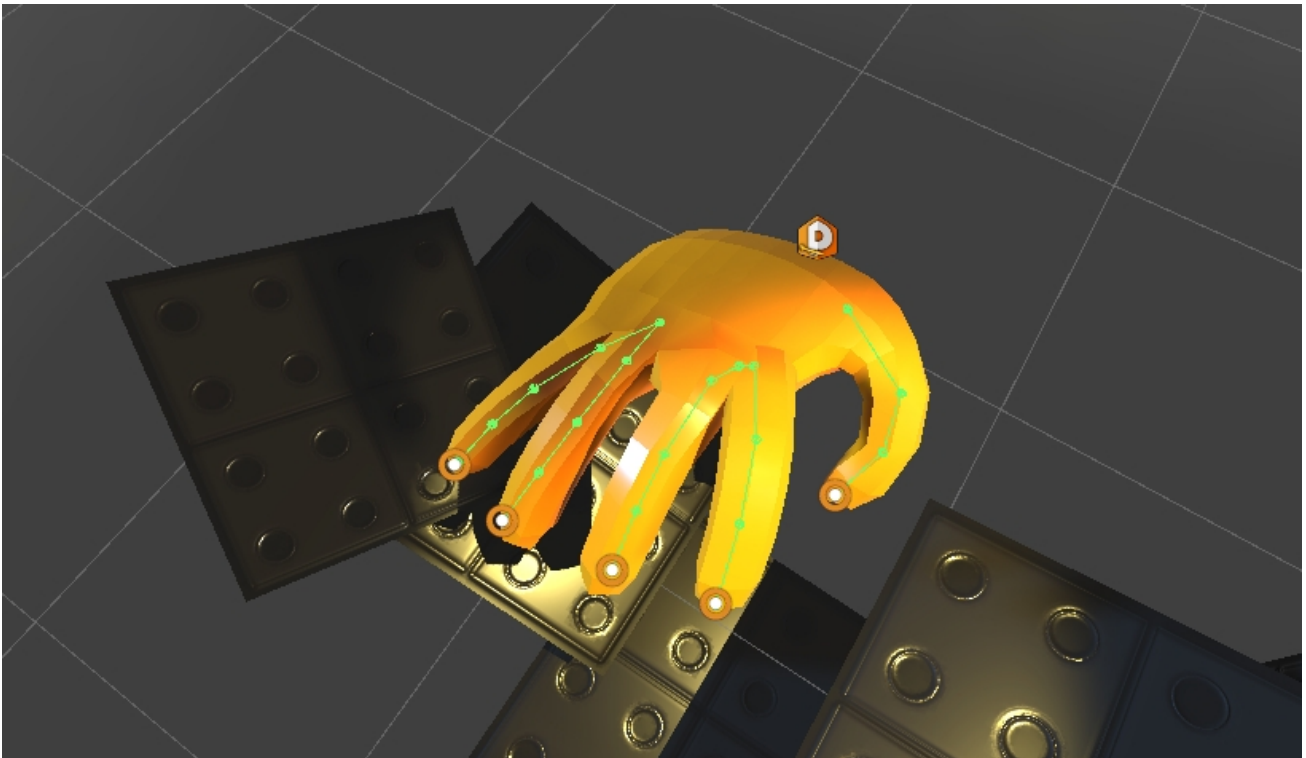
In **Properties** you can expect to find parameters that often times could be animated with code or via the IKP blender to alter the pose and interpolate between states, such as when you have a character picking up an item or reaching out for a ledge, etc. Programmatically those are accessed through the IKP API: `ikpInstance.SetProperty(...)`. More on the scripting later.

And now, after you have added all modules you want, you can use “**Quick Setup**” to quickly get the bone transforms into place, however sometimes that might not be enough. If the automatic finder fails, you will need to go to the *Setup* panel and manually link the references. Often failure might be caused by strange armature names that don't match with the *IKP bone names library*.

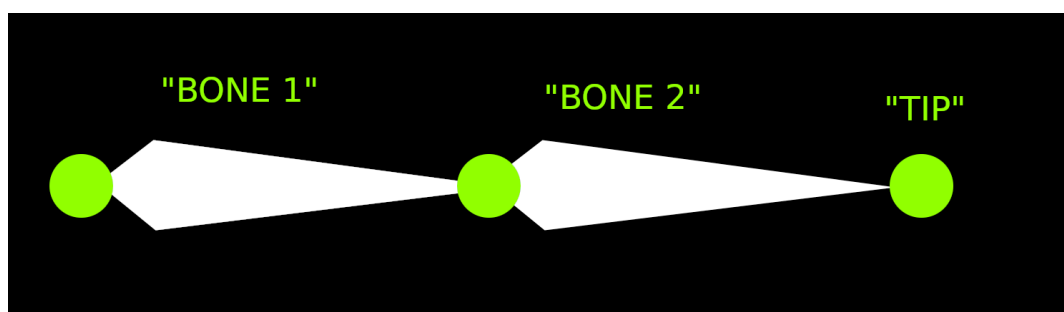
After you have successfully added all transforms, the red errors should disappear.

The Generic Limb Module

Generic Limb module can take any amount of bones with various lengths and make them align in the best way possible to reach a point in space.



Link the bones in the bone array in order from the root to the tip bone. With this module it's recommended to have an "end" or "tip" bone in the bone hierarchy. If you miss it in the model, adding a simple empty game object will do.



“Solver Iterations” (previously “Iterations”) specify the number of times the calculation tries to approximate the optimal bone positions. The higher the number - the more accurate the approximation.

“Is Leg” - If true, the tip of the limb will be oriented vertically, as if it attempts to step onto a surface.

“Stretchable” allows the limb to stretch (x0.5, x1, x2 etc..).

“Stretch Independent Of Weight” is for when you want to have zero weight on the limb (follows animation) but still maintain the overall stretch/scale of the limb.

“Stretch Always Reach” will stretch the end bone so that it reaches the exact position of the target.

“Is 2D” sets the generic limb to work in 2D space (for 2D games).

“Use Orientation” makes sure the bones follow a general direction when bending. If that option is turned off, the limb will behave more like a cable or a hose.

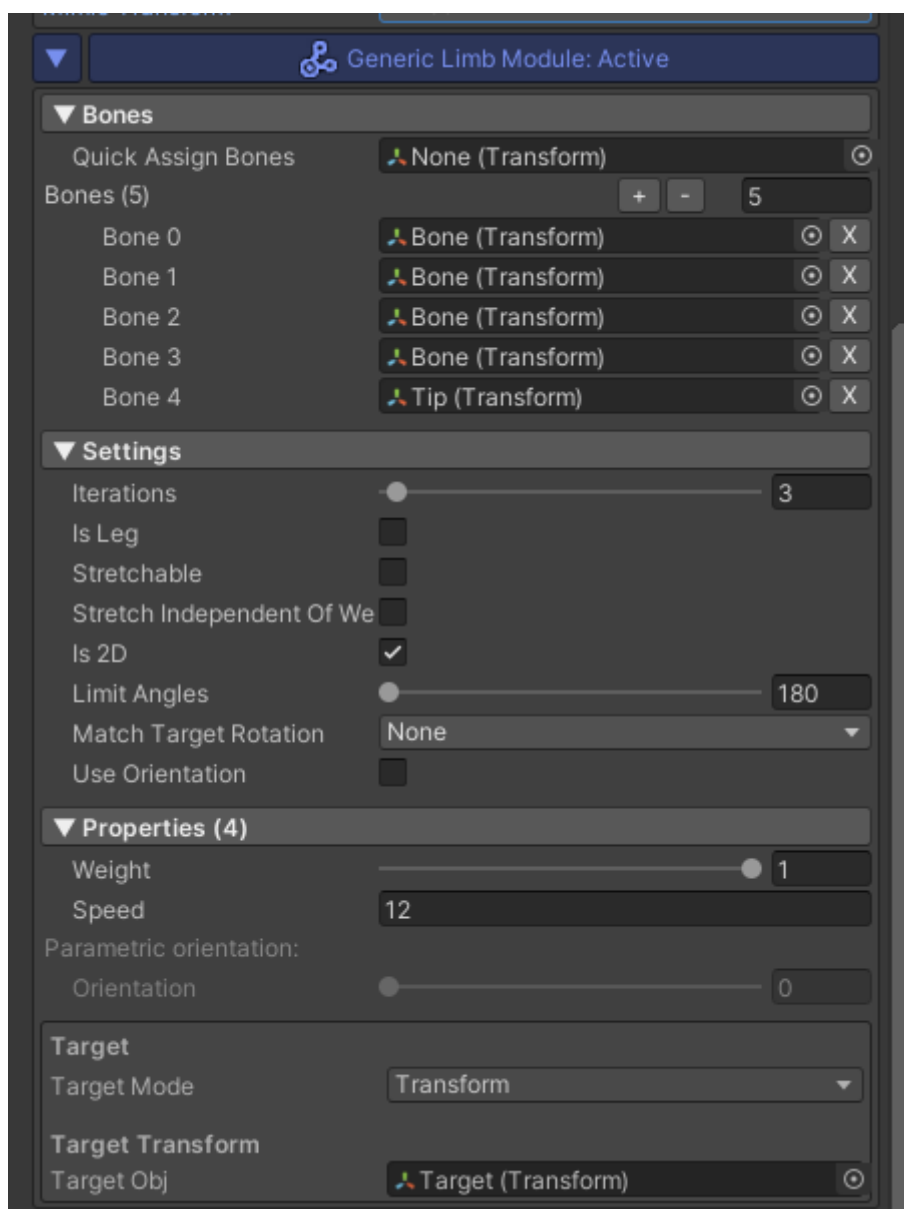
“Orientation Mode” selects between the two modes – parametric and positional.

- Parametric is a simple 360 arc around the origin-target axis.
- Positional takes in a list of points that define the shape of the orientation. This will also expose the “Chaikin Iterations” settings – which defines how smoothly the bones follow the orientation. Increasing this value could be quite taxing on performance so recommended value is 1.

“Weight” is the blending with the previous frame or animation.

“Orientation (Parametric)” is the percentage of a 360 angle (where 360 is represented as 1) to which the bones will gravitate.

“Speed” is how fast the limb reacts to change in the target.



Extension Methods:

```
void SetGenericLimbTarget (this IKP ikp, Vector3 position,
    Relative? relativeMode = null);
void SetGenericLimbTarget (this IKP ikp, Transform tr);
```



```
void SetGenericLimbTarget (this IKP ikp, IKPTarget ikpTarget);
```

```
void SetGenericLimbIterations(this IKP ikp, int amount)
```

Example (where “ikp” is an instance of IKP):

```
    ikp.SetGenericLimbTarget(new Vector3(1f,25f,0.5f));
```

The Head Module

The head module is responsible for the rotation of the character's head and neck. It requires 3 transforms – head, neck (optional) and chest bone.

The **look speed** (how fast does the character look at the target), **weight** (responsible for the blending between the animation and the IK pose) and **target** can be set through the *module panel*. Also another important property is the toggle option “**Has Neck**”, which lets the tool know whether or not the character has a designated neck bone.

“**Angle Limit**” – represents the maximum rotation angle of the head.

“**Track Target On Back**” – when that option is enabled, the head will constantly attempt to rotate in the direction of the target even if it's beyond the angle limit (when outside of the view angle, the head will ‘hint’ towards the target), resulting in a more believable behavior.

Extension Methods:

```
void SetLookTarget (this IKP ikp, Vector3 position,
    Relative? relativeMode = null);
void SetLookTarget (this IKP ikp, Transform tr);
void SetLookTarget (this IKP ikp, IKPTarget ikpTarget);
```

Example (where “ikp” is an instance of IKP):

```
ikp.SetLookTarget(new Vector3(1f,25f,0.5f));
```

This type of extension methods exist to make it easier to modify targets, without having to hold a reference to each module. They come as

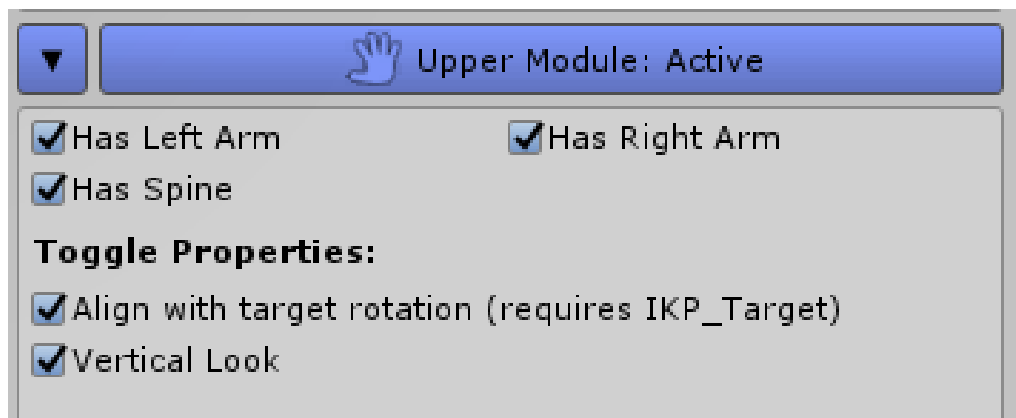
part of the modules themselves and are not found inside the IKP script, which is part of the modular approach.

The Upper Body Module

The Upper Body module is responsible not only for the torso, but for the arms as well. It allows the chest to assist the rotation of the head or the reach of the arms, or both. To work properly it requires the following transforms – *hips*, *spine**, *chest*, *left shoulder**, *left elbow**, *left hand**, *right shoulder**, *right elbow** and *right hand**. The bones marked with ‘*’ are optional.

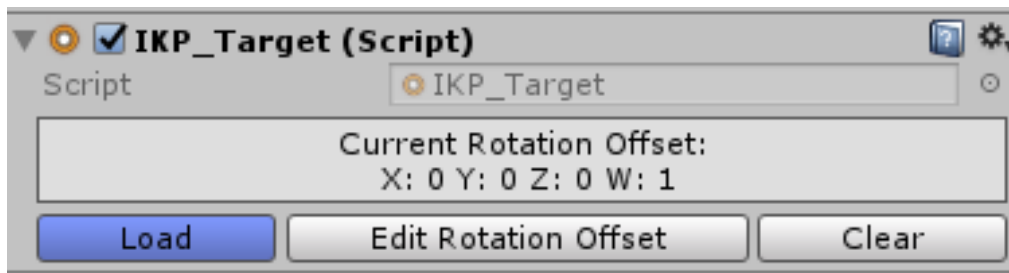
Note: *It’s recommended that the shoulder-elbow bone and the elbow-hand bones are roughly the same length.*

For example, if you disable the “**Has Left Arm**” (toggle) property, the tool will not require you to setup any bones associated with the left arm (Shoulder, Elbow, Hand) and won’t attempt to use it.



If your character doesn’t have a designated bone for the spine (between the hips and the chest), you can also uncheck the (toggle) property “**Has Spine**” so the tool won’t try to use it.

“**Align with target rotation**” will align the hands with the rotation of the target and if an **IKP_Target** component is found on the target transform, its rotation offset values will be applied.



“Vertical Look” will allow the chest to track the look target vertically.

“Forced Positioning” will attempt to correct any limb positioning error. Forced positioning is applied after the joint rotations but in some rare cases might lead to undesired mesh deformations.

Moving onto the **properties**, we have:

****“General Weight”*** is responsible for the weight of all body parts that are controlled by the upper body module and it multiplies all other listed weight properties.

****“Left Arm Weight”*** controls the weight of the left shoulder, left elbow and left hand. The same applies for the ***Right Arm Weight*** accordingly.

****“Chest Weight”*** determines how much the chest bone will assist the head rotation/the reach of the arms (this value is then multiplied by 0.65f inside the calculation in attempt to preserve the realistic behavior of the torso).

“Left Elbow Rotation” is used to control the direction at which the elbows are pointing, where 1 is furthest apart and 0 is the closest inward, with an angle between the two states of roughly 120°. The same applies for the ***Right Elbow Rotation***.

“Look Speed” is the speed with which the chest tracks the target rotation.

Each hand’s **target** can be set individually using the target interface in the *Module Panel* or by using one of the [utility functions](#) of the IKP.

“Max Chest Angle” – determines the maximum allowed angle at which the chest will attempt to track the target. It ranges from 0.1° to 179°.

“Track Target On Back” – which, when toggled on, will clamp the rotation of the chest to its maximum allowed angle (determined by “Max Chest Angle”), emulating an attempt to face the unreachable target (if it is in fact unreachable). Otherwise (when toggled off), once the target goes beyond the angle limit, the chest will align with the hips (facing forward).

“Forced Positioning” allows for more accurate positioning of the joints, however, its use sometimes might lead to unwanted deformations in the mesh.

Extension Methods:

```
void SetChestTargetMode (this IKP ikp,
    ChestTargetMode ct);

void SetHandTarget(this IKP ikp, Side side,
    IKPTarget ikpTarget);

void SetHandTarget(this IKP ikp, Side side,
    Transform transform);

void SetHandTarget(this IKP ikp, Side side,
    Vector3 position, Quaternion? rotation = null);
```

The Lower Body Module

The lower body module takes care of the character's legs. When both legs are enabled through their respective properties ("**Has Left Leg**" and "**Has Right Leg**") the module requires 7 bones in total – *Hips, Left Thigh, Left Knee, Left Foot, Right Thigh, Right Knee and Right Foot*.

Note: *It's strongly recommended that the thigh-knee bone and the knee-foot bone are the same length.*

Besides the aforementioned "Has Left Leg" and "Has Right Leg", inside the *Module Panel* of the lower body you can find the following set of properties and settings:

"Feet IK" – makes the feet react to the surface.

"Leg Raycasting" – makes the legs react to the surface, preventing clipping through objects.

"Forced Positioning" – will attempt to correct any limb positioning errors. It's applied after the joint rotations and in some rare cases might lead to undesired mesh deformations.

"Feet Length" – tells the length of the character's feet, which affects the feet IK calculation.

"Feet Offset" – this is the difference between the heel (touching the ground) and the position of the foot bone, responsible for the motion of the foot.

"Toe Height" offsets the "collision" point of the toes and the surface when "**Feet IK**" is enabled.

“Grounder Behavior” The job of the grounder is quite simple – it lowers the *hips bone* of the character to match changes in the underlying surface, based on the legs’ reach.

“Forced Positioning” (like for the upper module) allows for more accurate positioning of the joints, however, its use sometimes might lead to unwanted deformations in the mesh.

“General Weight” is responsible for the weight of all body parts that are controlled by the lower body module and it multiplies all other listed weight properties.

“Left Leg Weight” controls the weight of the entire left leg (thigh, knee, foot). The same applies for the **“Right Leg Weight”** accordingly.

“Left Knee Rotation” controls the direction in which the knee bends, where *0.5* (considered a “default” value) is facing forward relative to the *origin*, *0.75* is 90° outwards, *1 (and 0)* is 180° (backwards), completing a full circle around the thigh-foot vector. However, this property might vary for different character models. The same applies for **“Right Knee Rotation”**, only difference is the rotation direction (if it is counter-clockwise for the left leg, it will be clockwise for the right leg).

“Left Foot Rotation” rotates the foot around the thigh-foot vector, while with that simultaneously rotates the knee accordingly to achieve realistic motion. Giving the property a value of *0* will point the toes inwards to about 60° from the forward facing vector of the *origin*, where a value of *1* will mean a rotation of about 60° outwards. (As mentioned for the knee rotation, the same applies for the **“Right Foot Rotation”** with the difference being in the rotation direction).

“Leg Smoothing” has the role of controlling the speed at which the legs react to changes in the environment when **“Leg Raycasting”** is enabled. The higher the value, the faster the motion.

“Feet Smoothing”, similarly to leg smoothing - controls how fast the feet react to the underlying surface when **“Feet IK”** is enabled (higher value means faster motion).

“Grounder Weight” is the strength of the grounder effect.

“Grounder Reach” is the maximum drop of the grounder effect.

Extension Methods:

```
void SetLegTarget(this IKP ikp, Side side,  
    IKPTarget ikpTarget);
```

```
void SetLegTarget(this IKP ikp, Side side,  
    Transform transform);
```

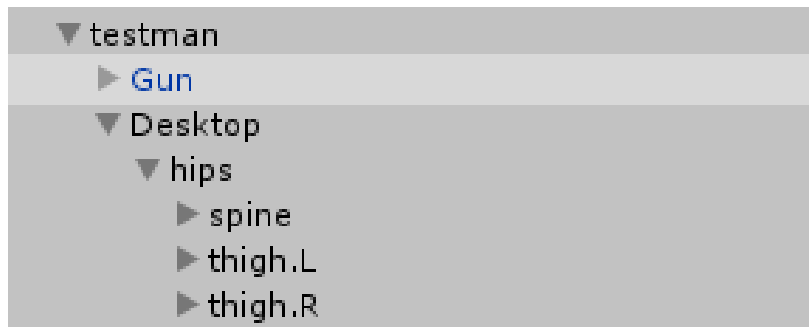
```
void SetLegTarget(this IKP ikp, Side side,  
    Vector3 position, Quaternion? rotation = null);
```

Example:

```
ikp.SetLegTarget(Side.Left, new Vector3(0f, 10f, 5f));
```

The Weapon Module

The weapon module is meant to help with adding proper weapon behavior, while with that, eliminating the update order problems that may occur with a custom script. The module requires a working Upper Body module. The **weapon transform**, although required for the module to work, is not mandatory during setup and it can be set through code in runtime. However, it's important to note, the weapon transform should NOT be a child of any bone used by the IKP (for example the hands). It's best if it remains a child only of the *origin* transform.



In the *module panel* there is the **“Weapon Pivot”** drop-down. It tells which hand the algorithm should use as a base/pivot for the weapon movement. You can select from left or right, and the according hand will be used (if it is setup in the Upper Body Module).

“Weight” tells how much the module moves the *weapon transform*. Lower weight will move the weapon less, to the point where the weapon transform is unaffected by the module at all, which will result in a state where the hands (of the Upper Body Module) follow the weapon transform, which can be now moved freely.

“Weight Ratio” determines how much is the weapon affected by the targeted hand pose of the *Upper Body*, and how much by the aiming algorithm, meaning a value of 0 would make the weapon be completely moved by the hand’s pose (selected by *Weapon Pivot*), where a value of 1 would aim the weapon first and then move the hands to the specified by the **IKP_Weapon.cs** script positions.

Tip: You can make the IKP character “hold” objects in hand (besides weapons) by having the Weapon Module’s **“weight ratio”** set to 0.

“Forward Offset” moves the origin of the weapon forward from 0, where the hand is the closest to its shoulder, to 1 – the maximum possible stretch of the arm.

“Aim Target” is the panel where you setup all the properties of the weapon target. Those properties (like all other *module panel* members,

except the toggle ones like “Has Neck/Arm/Leg/etc.”) can be set through code. The first thing you would want to do is tell the IKP what type of target would you like to use, and for that there are two options:

- **Target Inheritance** will use the Head modules’ target position (if there is a Head module).

- **New Target** will give you access to a panel where you can set independent properties, and the module will use those instead.

Extension Methods:

```
void SetWeaponMode (this IKP ikp, IKPAimTargetMode atm);  
void SetWeaponMode (this IKP ikp, IKPAimTargetMode atm,  
    Side side);  
void SetWeaponTarget (this IKP ikp, Transform transform);  
void SetWeaponTarget (this IKP ikp, IKPTarget ikpTarget);  
  
void SetWeapon(this IKP ikp, Transform transform);  
//Note that SetWeapon will update the weapon transform  
on the Weapon Collision Module as well.
```

The Weapon Collision Module

The weapon collision module is meant to make the held weapon interact with the world environment, preventing clipping through objects. For that purpose an IKP_Weapon component on the weapon transform is required. The Weapon Collision works well when paired with a Weapon Module, but can operate on its own as well.

The module requires a transform for the weapon, which when paired with a Weapon Module can be automatically inherited.

The Editor Simulation Module

Editor Simulation module allows you to test the IK movement inside the editor while making persistent changes to other module **settings**.

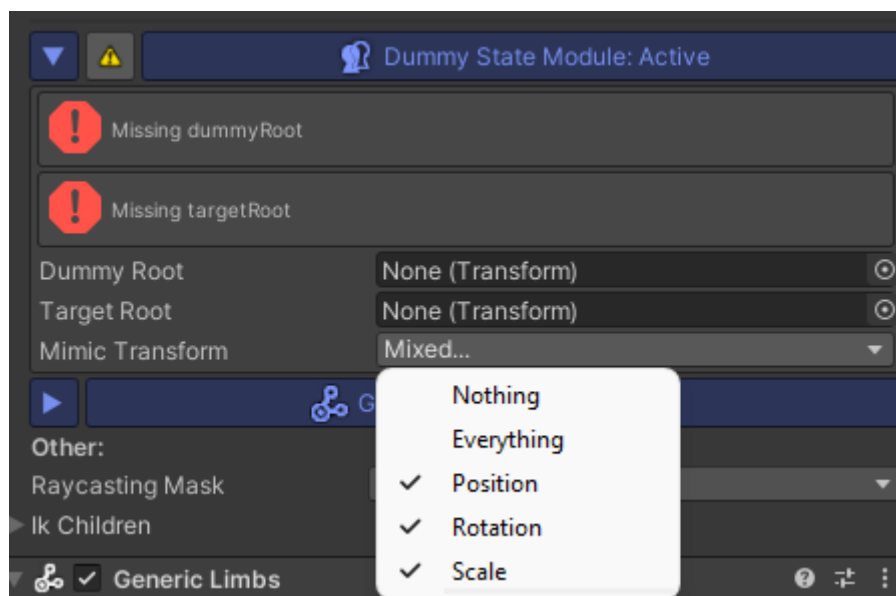
Note: Modifying **properties** during a simulation is not allowed.

The Dummy State Module

Dummy State module copies the transform data (position, orientation, scale) of a given source and its children over to the target (and its children, with respect to hierarchy) in the beginning of each update. This is useful for cases where you want to have a physically simulated body and interpolate between that and the IK pose. Such setup would of course require two separate armatures – one with physics and one with IK.

Put the source root you want to copy from in Dummy Root and the armature you want the source data copied over to – in Target Root.

You can additionally choose what to mimic with the Mimic Transform mask.



IKP_Weapon script and weapon setup

This script is designed to allow the Weapon Module to know where the hands should be located on the weapon. Simply attach it to the root transform of your weapon and add in the transforms for left and right hands (it is not required to add both). It's even better if those transforms contain the **IKP_Target.cs** script on them.



Note: It's important that the game objects representing the hand positions are NOT outside the weapon's root transform.

“Forward Offset” variable multiplies the one found on the Weapon Module, so it is best if you keep the Weapon Modules' *“Forward Offset”* value to 1, and you use this weapon-specific one instead.

Setting Targets & Properties through code

For an overview of the custom types that each method takes, refer to the [IKP Types Library](#).

About the targets:

The modules come with extension methods that allow setting targets with an instance of the `ikp`, rather than a reference to each module component individually. Those methods are usually found in a `cs` file named with the “Extend” suffix. For the primary modules, you can also find their extension methods listed in their respective paragraphs above: [GenericLimb](#), [Head](#), [Upper](#), [Lower](#), [Weapon](#).

Example: (where `ikp` is an instance of `IKP.cs`)

```
Ikp.SetLookTarget(new Vector3(1f, 2f, 3.5f),  
    Relative.World);
```

About the properties:

While working with IKP it's important that you are able to manipulate properties on the fly from within your code. The IKP component has methods that allow you to change the weight and rotation properties of a module without needing to hold a reference to each component individually or having to deal with tons of additional code. You may want to increase or decrease the influence which a specific module has during gameplay or rotate the elbows/knees/feet/etc. of a character. Each module has a different amount of properties. Some modules may even have only one or two such properties. That's where the utility functions of the **IKP.cs** components come in handy.

When you first add and setup a module it will probably have default weight values set to 0, and even if you set a target, the IKP module will still have no effect on your character. So whenever you are working with the IKP through code at runtime, you will most probably need to set a few property values.

Assuming you already have a reference in your code to an IKP instance, (*IKP ikpInstance*), a property value is set as follows:

```
void SetProperty (int property, float value,  
                 string moduleSignature);  
void SetProperties (float[] values,  
                  string moduleSignature);
```

In the first method, the first parameter is the **index/id of the property** you want to set, then it's followed by a float **value** and finally – by a **module signature**.

Module signature represents the specific module you want to set properties of. You can find it in the Module Assistant on the module linkers page. It's recommended that you use *ModuleSingature* instead of string literals:

- `ModuleSignatures.GENERLIC_LIMBS,`
- `ModuleSignatures.HEAD,`
- `ModuleSignatures.UPPER_HUMANOID,`
- `ModuleSignatures.LOWER_HUMANOID,`
- `ModuleSignatures.WEAPON,`
- `ModuleSignatures.WEAPON_COLLISION,`
- `ModuleSignatures.EDITOR_SIM`

Now, knowing that each module has a different set of properties, it's important to understand what those properties are and what is their order. For that you can refer to the [Module Properties Table](#), listing all enums under the same name for each module:

(Keep in mind, that although the property enums share the same name, they are not interchangeable and that is the reason why the property id argument of `SetProperty(...)` method is an integer)

```
ikp.SetProperty(int propertyId, float value,  
                string moduleSignature);
```

Example: *(where ikp is an instance of IKP.cs)*

```
ikp.SetProperty((int)IKPModule_Head.Property.Weight,  
                1f, ModuleSignatures.HEAD);
```

All of the above-mentioned properties have a “cache” variable referring to their index, so you don't have to do `(int)IKPModule_Head.Property.Weight` and you can just do `IKPModule_Head.p_weight`. All of those variables follow a simple convention:

“p_” + (the property name, starting with a lower case)

So `(int)Property.Weight` would be `p_weight`. That way you avoid casting to int each time you want to use the property's index and you also save yourself some code space.

Now the above example would look something like this:

```
Ikp.SetProperty(IKPModule_Head.p_weight, 1f,  
ModuleSignatures.HEAD);
```


For the module signatures and other useful information you can look up the **Linker Assistant** page of the **Module Assistant** panel. It displays all currently imported modules alongside their linker values such as signature, type name, update order, etc.

Toggle a module:

Some times you may need to toggle a module on or off. Do that using the method:

```
ikp.ToggleModule((string) moduleSignature, (bool?)  
state = null);
```

Example: `ikp.ToggleModule("GenericLimb");`

Or alternatively:

```
ikp.GetModule(signature).SetActive(state)
```

Module Properties Table

IKPModule_GenericLimb - (*enum*) **Property:**

Weight, Speed, Orientation

IKPModule_Head - (*enum*) **Property:**

Weight, LookSpeed

IKPModule_UpperBody - (*enum*) **Property:**

GeneralWeight, LeftArmWeight, RightArmWeight,
ChestWeight, LeftElbowRotation,
RightElbowRotation, LookSpeed

IKPModule_LowerBody - (*enum*) **Property:**

GeneralWeight, LeftLegWeight, RightLegWeight,
LeftKneeRotation, LeftFootRotation,

RightKneeRotation, RightFootRotation,
GrounderWeight, GrounderReach, LegSmoothing,
FeetSmoothing

IKPModule_Weapon - (*enum*) **Property:**

Weight, WeightRatio, ForwardOffset

IKPModule_WeaponCollision - (*enum*) **Property:**

ReactionSpeed

The IKP Blender

The IKP Blender is designed to help you blend your IKP pose with the currently playing animation. It animates the properties of a module in run time (such as weights, rotations, etc.). To set it up you will simply need to add the IKP_Blender.cs component to the game object containing the IKP. After that you will need to add an *IKP Blend Machine*.

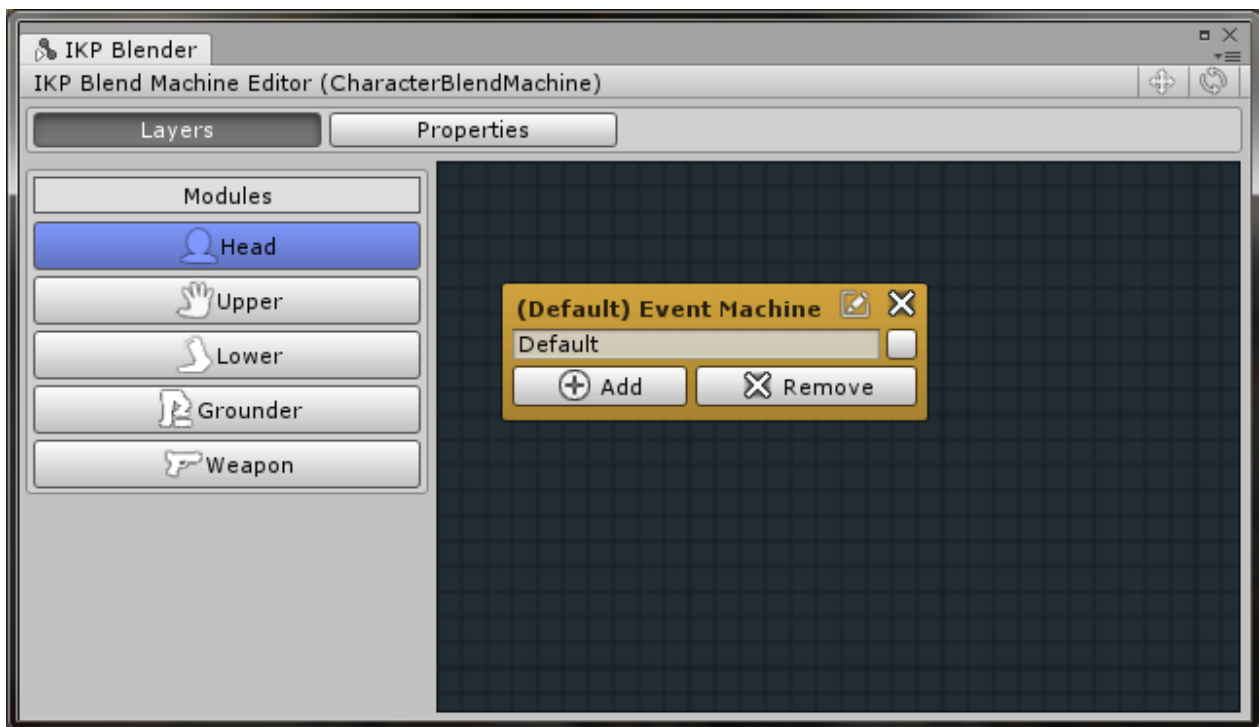
The Blender uses event calls to direct the *blend animations*. Given that you have an instance of the IKP_Blender (for the purpose of this example let's name that instance *blender*), you can use *blender.CallEvent("Event Name");* to start an event from the *Blend Machine*. Calling an event will instantly interrupt the currently playing blend animation and will start the first found event with the same name as the call. If you want to call an event inside an Update method, you can use *blender.CallEventOnce("Event Name");* however, using this more than once per blend machine might lead to unpredictable behavior. That's why it's highly recommended that you call events once (outside of loops and updates) with *CallEvent(...)* only when a certain condition is met.

Other useful methods of the IKP_Blender are:

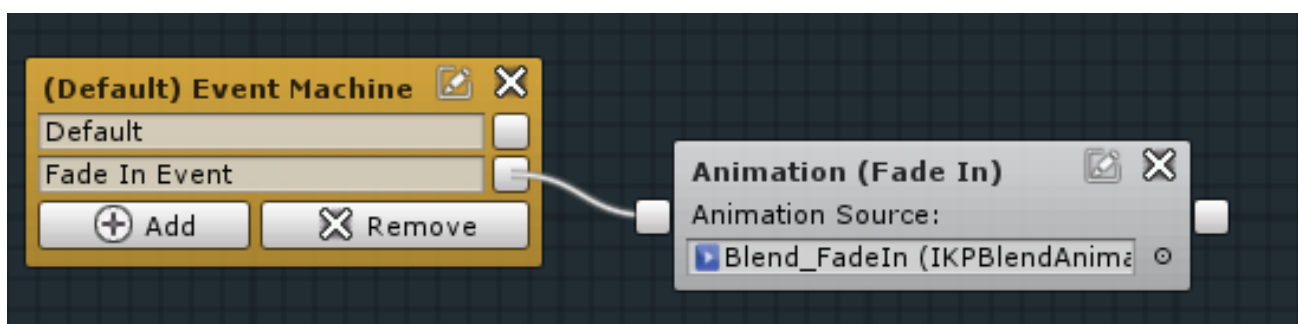
```
bool HasBlendMachine(); //returns true if a blend
machine has been assigned to the Blender
IKPBlendMachine GetBlendMachine(); //returns the
current ikp blend machine
string GetCurrentState(); //returns the current
state, result of the last called event
```

Using the Blend Machine Editor

Creating a blend machine is as simple as right clicking anywhere in your project and selecting **Create > IK Plus > New Blend Machine**. Then a new Blend Machine will be created and you can click “Edit Blend Machine” to open up *Blend Machine Editor*.



On the left of the window you will see a panel with modules. Each menu item is responsible for animating the according module's properties. By default each module *field* has one “*Event Machine*” node with one “*Default*” event. You can create new events by clicking the *Add* button on the Event Machine node.

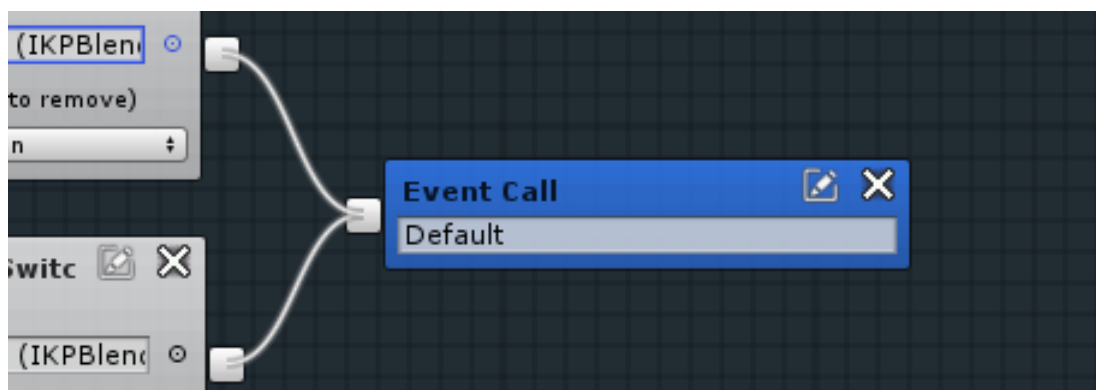




Once you have the desired events in, you would want to connect them with blend animation nodes. Right click and add a new *Blend Animation*. That will create an empty Blend Animation node then you will have to drag in the needed *IKP Blend Animation* from your project to the empty slot.

If you, however, happen to need some properties from the selected animation ignored for the specific node, you can add them to the ignored properties list by clicking “Add Property Exception”. Note that only active properties will show up. If the current animation doesn’t utilize the a property you won’t be able to add it to the list.



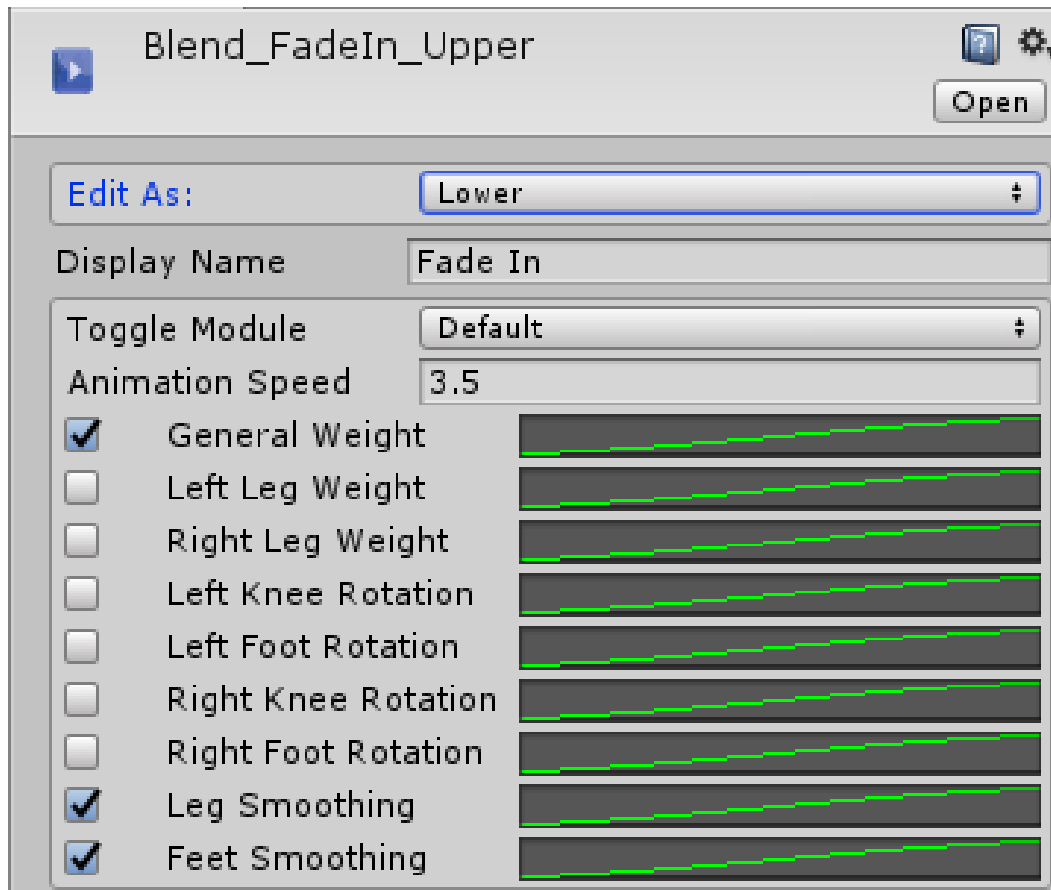
There is also the “Event Call” node, which when reached will call the event defined inside it.



You can also **rename** your nodes by clicking the edit icon  in the top-right and you can **delete** a node by clicking the X icon . None of those actions can be undone.

Creating blend animations

Creating a blend animation is done by going to your project panel and choosing Create > IK Plus > New Blend Animation.



Inside the inspector of the Blend Animation you can select the module view you want to work with (Head, Upper, Lower, etc.) from the drop-down **“Edit As:”**. Each view will present you with different interpretations of the animation values, matching the properties of the according module.

You can pick **“Display Name”** for your animation, which will show up as your animation node’s name.

Animation behavior values:

You can select if you want to toggle a module at the start of the blend animation with “**Toggle Module**”, which offers you to either “*Enable*”, “*Disable*” or leave it “*Default*” (leave-as-is).

“**Animation Speed**” is a multiplier for the speed at which the animation plays (default is 1).

Each property value is made up of 3 components – a toggle, a property name display and an animation curve.

- The **toggle** enables or disables the behavior of that value in the animation; meaning when it's *off*, the animation of the value will not affect the specified module property.
- The **display name** is there to let you know which specific module value will be affected. It cannot be changed.
- The **animation curve** represents the change in the value of the property over time.

IKP Types Library:

Here are the main types you will come across. To use them you will need to declare ***using IKPn;*** in the beginning of your C# script.

Standard enums:

enum BodyParts

Head, Neck, LeftShoulder, LeftElbow, LeftHand,
RightShoulder, RightElbow, RightHand, Chest, Spine,
Hips, LeftThigh, LeftKnee, LeftFoot, RightThigh,
RightKnee, RightFoot

enum IKPTargetMode

Position, Transform

enum Relative

World, Object

enum Side

Left, Right

Upper Body Module enums:

enum ChestTargetMode

None, LookTarget, HandsReach, Combined

Weapon Module enums:

enum IKPAimTargetMode

NewTarget, TargetInheritance

Also look up [Module Properties Table](#).

Important Custom Classes:

class **TransformData** (from SETUtil)

Fields:

```
Vector3 position;  
Quaternion rotation;
```

Constructors:

```
TransformData(); //default constructor  
TransformData(Vector3, Quaternion); //constructor from  
Position & Rotation  
TransformData(Transform); //constructor from Transform
```

Methods:

```
void Set (Vector3 position, Quaternion rotation);  
void Set (Transform tr);  
void Set (TransformData trdt);
```

class **IKPTarget** (from IKPn)

Fields:

```
Vector3 targetMode;  
Vector3 targetPos;  
Quaternion targetRotation;  
Relative relativeTo;  
Quaternion? targetRot;  
Transform targetObj;
```

Constructors:

```
IKPTarget (); //default constructor  
IKPTarget (Transform transform); //From transform  
IKPTarget (Relative relativeTo, Vector3 pos, Quaternion?  
rot = null); //constructor from Position & Rotation
```

```
IKPTarget (IKPTargetMode tgtMode, Relative relativeTo,  
Vector3 pos, Quaternion? rot = null, Transform tr = null);
```

Methods:

```
TransformData Get (Transform pivot); //pass the origin as  
pivot parameter  
Quaternion? GetRotation (); //get only the rotation of the  
target
```

Dictionary

“Module component” is the component that has been added in the inspector under the IKP (Runs module logic).

“Module panel” is the graphic interface inside the IKP component. From it you can control all the important properties and settings of a module or enable and disable it.

“Weight” refers to a property that blends the specified character body part between the animation that is currently playing and the pose, generated by the module.