IK Constructor

Easy Inverse Kinematics for simple solutions.

Approach

IK Constructor is a new Inverse Kinematic solution for Unity Engine. Although it is not as flexible as CCD of FABRIK methods, it can be used for many tasks. It just makes things work in its own way. First of all, it does not contain a global solver. Instead, it solves a situation using multiple local solvers. Each local solver does a single thing – aims the target. To understand pros and cons of this approach consider a game of chess. Every chess player knows - sometimes, it needs to make a bad chess piece exchange to win the game. Global IK solvers aware of that. IK Constructor does not, so, it is less flexible. On the other hand, global solvers using multiple passes to adapt the rig, and in some (rare) cases precision is not satisfying, or there is a multiple concurrent poses, and it Is hard for the solver to choose between them. IK Constructor does all moves with absolute precision (minus floating point operation inaccuracy) and requires only a single computation cycle for each frame. There is no concurrent poses problem, because of absence of multiple solutions. You will see that clearly as you read the rest of this documentation.

Basics

Basic description

IK Constructor is a set of scripts – or inverse kinematic templates and each of them does its own thing. Every script can work right in the Unity Editor window without pressing Play button. Scripts can be stacked upon each other to form any kinematic chain you want. Be aware that because of simplicity of approach to the IK solving problem this constructor cannot drive ANY chain. It was designed to drive only simplest of IK-chains. Still, practice proves it is enough for the most of simple game designs. Also, additional scripting can enlarge area of application.

Initialization

Any component initializing automatically as its Origin field assigned. In the initialization process component stores some or all local parameters of assigned transform into its internal memory. During following updates all computations will be done using that data. After component was initialized successfully, it starts to drive assigned game object, and user loses control of game object's transform. In some cases a secondary initializing must be performed. In order to do that:

1. Turn component off in the inspector – it will lose control over a transform.
2. Change transform as you need.
3. Press "Set Initital State" button in the bottom part of component.
4. Turn component back on.

If you need to initialize a component from script – call public Init() function or any of its variations.

Updating

By default every component updating automatically in its own LateUpdate() function, but there is a possibility to turn an automatic updates off. To do that trigger AutoUpdate button in the inspector or reset a property with the same name from script. Every component have a ManualUpdate() method, using which you can update it from your custom script. This is useful for precise IK-chain update control or in case, when you need to insert additional computations in the middle of IK-chain.

Visualization

Each component contains a debug visual, representing it in both Play and Edit modes of the Unity Editor. With this ability you can see it with no geometry assigned and without selecting it in the scene. This option can be turned off by triggering DrawVisuals button in the inspector or can be triggered by writing corresponding property from script.

Inverse and Forward Kinematic

Any component can work in Inverse or Forward kinematic mode. Current mode is defined by “Kinematic” parameter. By default an Inverse kinematic mode is set. In addition you need to set a game object as target in the “Target” field. After that is done a component will become active and will try to reach the target and measure angle/position relative to its default position/orientation. In order to choose Forward kinematic mode just switch a “Kinematic” field to “Forward” – target object will be ignored and you will be able to set angles/position manually and component will follow those values.

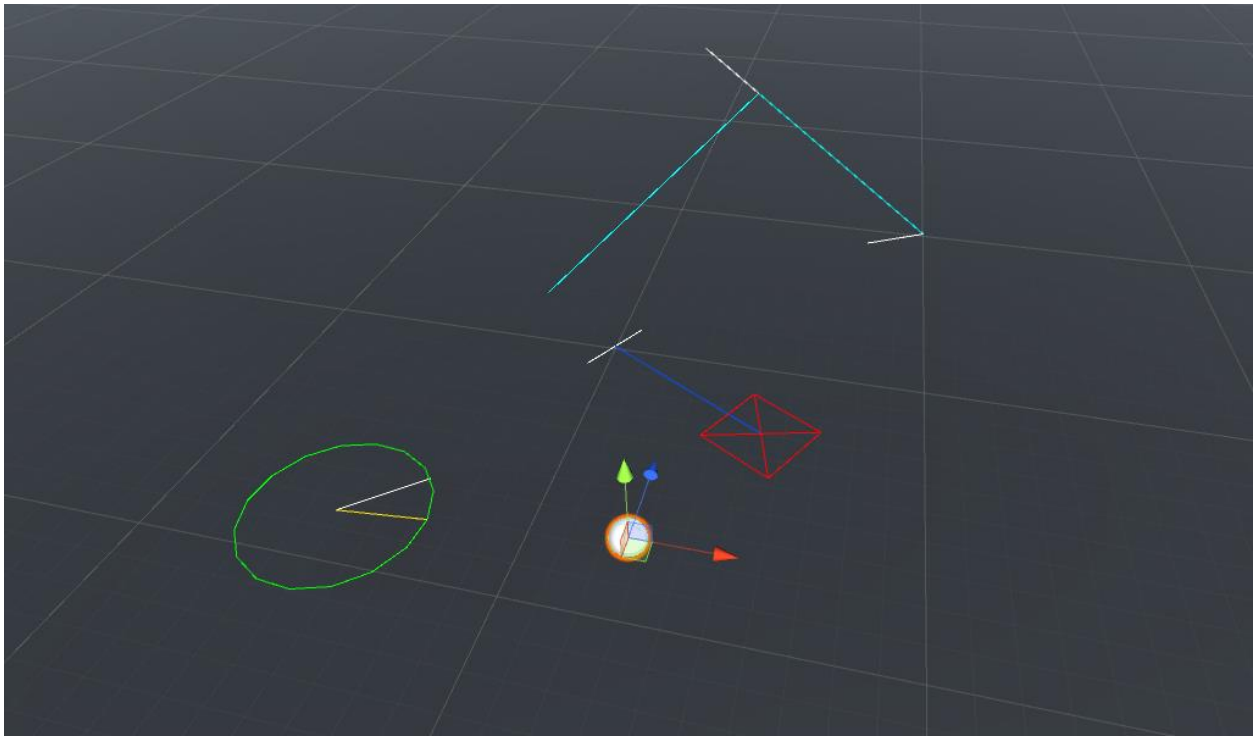
Synchronization of components

There is a way to synchronize two or more components in the way that second will repeat moves of the first. Components, which support that mechanism, contain two additional fields. Just assign a reference of the first component into field “Master” of the second. Note that slave will follow its master only in forward kinematic mode. Also, in slave mode user can no longer drive component manually. Master can work in any mode, even being slave of another component. Additional field “Master Source” defines what kind of value a slave will use as its primary. The only limitation is that master must be updated before its slaves.

Making simple IK-objects

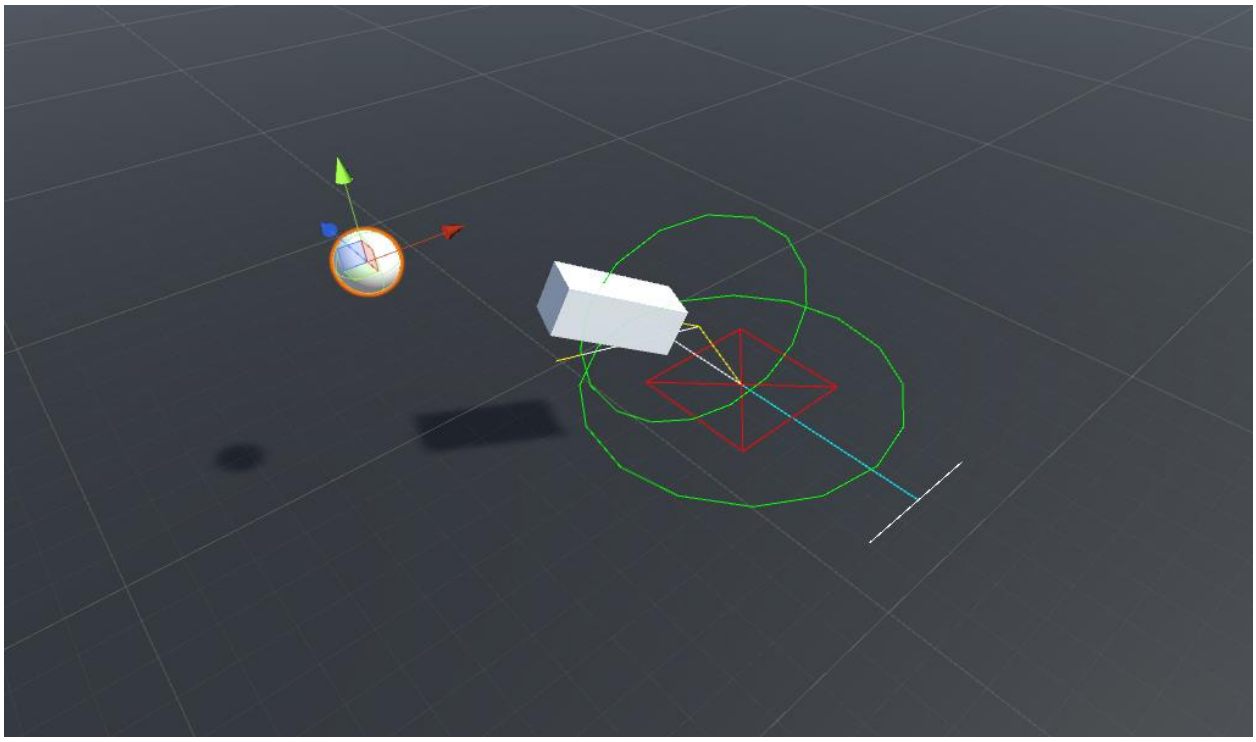
To Create a simple IK-object you need:

1. Select an existing game object or create a new one (even empty will do).
2. Drag any IK-component script to it. (IKAxis.cs, IKMover.cs, IKArm.cs)
3. Set kinematic mode you need (Inverse by default)
4. Drag this game object (or another one, you want to drive) in the “Origin” field. At this moment an automatic initialization will occur, and current transform data will be stored inside component as default.
5. If an Inverse mode was chosen, you need to drag a target object into “Target” field. After that object will instantly move/turn to the target and will start following it.



Creating an IK-chain

1. Create a few game objects and attach them to each other to form a hierarchy.
2. Set their relative positions and orientations as you need.
3. Add corresponding IK-scripts to drive each object. It is recommended to place all IK-scripts on a parent object for easier navigation and to control script execution order. Sort your IK-scripts so that (from top to bottom) first controls a parent, and last controls a lesser child. This is important for the end result.
4. Setup each component as you need.
5. Its ready to use right in the editor! All components will perform their update in top-to-bottom order.



Detailed description

Computation pipeline inside a component

Any component contains a computation pipeline which is defined by stages. Presence of each stage is individual for each component. Lets describe each stage and their order in common.

1. Solver stage. In inverse kinematic mode calculates an angle(s)/position(s) to reach its target and stores it in “TargetValue” field. In forward kinematic mode “TargetValue” field is left untouched to let the user define its value manually. In slave mode “TargetValue” is taken from another script, which was previously set in the “Master” field. “Master source” field defines what kind a value will be taken.
2. Converter stage. Takes “TargetValue” field, converts it, and places into ”AlteredValue” field.
3. Limiter stage. Clamps an AlteredValue within user defined range, if active.
4. Animation stage. Interpolates a “CurrentValue” field to match an AlteredValue instantly, with constant speed, or with acceleration. In addition animator updates flags, indicating end of animation, and target reaching.
5. Setting pose stage. Uses “CurrentValue “ field to set a new pose for drivable object.

Solver stage

Each component has its own individual solver, whose functioning is defined by a components purpose. If you need a complete details of what and how it does it see a method named “Solve<something>()” inside a component’s source code.

If solver returns an angular value, it always lies in [-180,180] degrees range. In forward kinematic mode, solver not active and user can set any angle, even unwrapped.

Converter stage

Takes a TargetValue and converts it by formula: AlteredValue = (TargetValue + PreOffset) * Gain + PostOffset. Parameters PreOffset, Gain and PostOffset will be seen in the inspector only if component supports this stage.

Limiter stage

If limiter works with linear value: minimum must be less or equal to maximum at all times. Making “dead zones” is not supported. In case you need one-sided limit just write “Infinity” or “-Infinity” into corresponding field.

If Limiter works with angular value: minimum and maximum form and arc, which goes counterclockwise from minimum to maximum angles specified. One-sided limiting not supported.

Animation stage

If component supports an animation stage it has two additional fields:

1. “Speed” – speed of motion/rotation
2. “AccelerationTime” – time, during which, animation value will be uniformly accelerated.

Animator works in one of three possible modes of operation, choosing them automatically in dependency of how Speed and Acceleration Time fields are set:

Animation mode	Condition
Instant	Speed=0
Constant speed	Speed>0, AccelerationTime=0
Acceleration ramp	Speed>0, AccelerationTime>0

1. Instant mode – Current value instantly sets as Altered value. No animation.
2. Constant speed – Current value interpolates towards an Altered value with speed specified.
3. Acceleration ramp – If overall travel distance allows, Current value changes with acceleration, then goes with constant speed, then decelerates.

Component description

IKAxis

IKAxis component it is a rotation axis which acts in the following behavior:

1. It rotates around its local “Y” axis (green gizmo rod)
2. Zero angle stays on local “Z” axis (blue gizmo rod)
3. Stores local orientation of supplied “Origin” transform.
4. Takes “Origin” position into account during computations, but does not control it.
5. It has a special feature called “Slide Compensator”. It acts as if an axis was shifted to the left or right along its local X axis. Usable if next object in IK chain is shifted away and you need precise targeting. (See tutorial for more details).

Axis parameters:

1. TargetAngle – current azimuth to the target.
2. AlteredAngle – TargetAngle passed converter and limiter stage.
3. CurrentAngle – current angle, controlled by animator.
4. IsFinished – Set to true, when animation finished, false otherwise.
5. IsTargetReached – Set to true, if animation is finished and “Origin” was able to turn directly to the target.
6. Kinematic – sets primary mode of operation. Inverse and Forward modes supported.
7. Master – A reference to master in case this object works as slave. Ignored in inverse kinematic mode. Only an Axis script can be set here.
8. MasterSource – Defines what master’s fiend slave will take as its TargetAngle.
9. Origin – A reference to game object’s transform to control using this script
10. Target – A reference to game object’s transform to turn Origin to.
11. Slide Compensation – Control value for axis shifting feature. Set to zero, if not used.
12. PreOffset, Gain, PostOffset – angle converter parameters.
13. Limits, Minimum, Maximum – angle limiter parameters.
14. Speed, AccelerationTime – animator parameters.

IKMover

Mover simply does, what its name stands for – moves. It acts in the following behavior:

1. Moving of the “Origin” transform goes along its local Z axis.
2. Stores local position of the “Origin” during initialization.
3. “Position” (or scalar distance) is measured relative to that position.
4. Local rotation taken into account, but Mover doesn’t control it.
5. Mover has a special feature, called “Distance Keeper”. You can always keep a certain distance to the target object instead of trying touching it. There is a different ways of how to measure that distance.

Mover parameters:

1. TargetPosition – scalar distance away from its default (initialized) position.
2. AlteredPosition – TargetPosition passed converter and limiter stages.
3. CurrentPosition – current position, interpolated by animator.
4. IsFinished – Set to true if animation is finished (AlteredPosition is equal to CurrentPosition), false otherwise.
5. IsTargetReached – Set to true if “IsFinished” is true, and if an AlteredPosition wasn’t limited during limiter stage.
6. Kinematic – Sets primary mode of operation.
7. Master – Holds a reference to a master script. If set, this mover becomes a slave. Only another mover script can be set in that field.
8. MasterSource – Which data to read from master to assign to TargetPosition.
9. KeeperMode, KeepDistance – distance keeper parameters. Keeper becomes active if KeepDistance is greater that zero.
10. PreOffset, Gain, PostOffset – converter paramaters.
11. Limits, Minimum, Maximum – limiter parameters.
12. Speed, AccelerationTime – animator paramaters.

IKArm

This component controls a hierarchy of 3 objects, formed as arm. Arm tries to reach the target within its end. If target is closer than the sum of both arm's "bones" length, the arm is bending in its elbow point. Arm behaves like this:

1. Arm's base turns around its local X axis (red gizmo axis).
2. X axis for the base always match X axis for an elbow.
3. Elbow always bends towards local Y axis (green gizmo axis).
4. Elbow object is away from base by "Origin Length" meters.
5. Endpoint object is away from Elbow object by "Elbow Length" meters.
6. Arm's base position takes into account but not controlled.
7. Arm's endpoint orientation is not controlled either. (See appendix 1)

Parameters:

1. Target Origin Angle – angle of rotation between base default orientation and a current one.
2. Target Elbow Angle – angle between elbow current orientation and base default.
3. IsTargetReached – true, if arm's endpoint is touching a target's projection on a plane in which arm works on.
4. Origin Length – distance between Origin and Elbow objects.
5. Elbow Length – distance between Elbow and Endpoint objects.
6. Kinematic – Primary mode of operation.
7. Origin – Base, or shoulder of an arm.
8. Elbow – Elbow of an arm.
9. Endpoint – Arm tries to touch the target with this object.
10. Target – Object, chosen as target for this arm.

Appendix 1. Who drives what

This table demonstrates how components controlling their transform's parameters. "Reads" means, that script is taking that field into account but never changing it, "Controls" means changing value each update, "Ignores" as it stands.

IK script	Object	Position	Rotation
IKAxis.cs	Origin	Reads	Controls
IKMover.cs	Origin	Controls	Reads
IKArm.cs	Origin	Reads	Controls
	Elbow	Controls	Controls
	Endpoint	Controls	Ignores

Appendix 2. Mover's distance keeper stage

To activate distance keeper just set a "Keep Distance" field to a positive value then set a "Keeper Mode" field to one of values from table below. Table explains how mover component measuring its distance to the target object.

"Keeper Mode" field	Description
Volumetric	True 3D distance between origin and a target.
Plane XZ	Target is projected on local XZ plane then distance is measured between that point and origin. Mover behaves as Target's Y position doesn't matter.
Plane YZ	Same as previous, just another plane is used.