GENERIC IK

Easy Kinematics simulation

Generics Studio 2017

Contents

Introduction

Generic IK has in version 1.1, 3 solvers for Inverse Kinematics, and 1 solver for Forward Kinematics.

To get up and running with Generic IK, there are 2 options:

- Using the included InverseKinematics.cs script
- Calling the Solver manually via a script

And we will go through them in the next Chapter.

Solvers

There are 3 main IK solvers in the package:

- Cyclic Descend Solver
- Fast Reach Solver
- Directional Swing Solver

And a FK solver for chain simulations called: Chain Kinematics Solver

Cyclic Descend Solver

Goal: Reach a target

Pros: utilizes Motion Freedom algorithms

Cons: Requires a little more iteration count to achieve IK for animated characters

Most suitable for: Reaching a target while applying Motion Constrains

Fast Reach Solver

Goal: Reach a target

Pros: very accurate, very few iterations

Cons: does not utilize Motion Constrains

Most suitable for: Reaching a target as quick as possible disregarding motion constrains

Directional Swing Solver

Goal: Orients a chain of joints in such a way that the local axis of the End Effector, Points at a target

Pros: fast, utilizes Motion Freedom

Cons: -

Most suitable for: aiming behaviors, look at behaviors, swing at behaviors

Chain Kinematics Solver

Goal: Apply physics to a chain of joints

Pros: fast, utilizes Motion Freedom

Cons: does not detect collisions with other GameObjects

Most suitable for: simple rope simulations, Tails, chains with physics

The Inverse Kinematics Script

The InverseKinematics.cs script is meant to sum up all solvers functionalities (except the Directional Swing, reasons here) in 1 pretty UI, you do not necessarily need it in your game.

OnEnable in edit mode, the script will detect the rig type of the model and draw the inspector GUI based on 1 of 2 possibilities:

- Generic Rig UI
- Humanoid Rig UI

Incase of Generic Rig UI, the script will show you a chains array that you can resize to use with the 2 reach IK solvers and a final array for Chain Kinematics usage.

Incase of Humanoid Rig UI, the script will build 4 limb chains. In Addition, you will also be given a chains array parameter similar to the previous case, for additional usages.

Using Generic IK via its API

An Alternative way of using Generic IK, is simply by creating a “Chain” variable, feed it with info in the inspector and calling the solvers in your code

Note: You must include the namespace ***Generics.Dynamics*** first.

Examples

```
public Core.Solvers solver;

public Core.Chain rArm, lArm;
public Core.Chain rLeg, lLeg;

public Core.Chain[] otherChains;
public Core.Chain[] lookAtChains;
public Core.KinematicChain[] otherKChains;
```

```
private void LateUpdate()
{
    switch (solver)
    {
        case Core.Solvers.CyclicDescend:
            CyclicDescendSolver.Process(rLeg);
            CyclicDescendSolver.Process(lLeg);
            CyclicDescendSolver.Process(rArm);
            CyclicDescendSolver.Process(lArm);
            for (int i = 0; i < otherChains.Length; i++) CyclicDescendSolver.Process(otherChains[i]);
            break;
        case Core.Solvers.FastReach:
            for (int i = 0; i < otherChains.Length; i++) FastReachSolver.Process(otherChains[i]);
            FastReachSolver.Process(rLeg);
            FastReachSolver.Process(lLeg);
            FastReachSolver.Process(rArm);
            FastReachSolver.Process(lArm);
            break;
    }

    for (int i = 0; i < otherKChains.Length; i++)
    {
        ChainKinematicSolver.Process(otherKChains[i]);
    }
}
```

Note that you do not need a “solver” variable. Simply if you want to use Fast Reach for your chain for example, just call the FastReachSolver.Process(chain) function.

Now there is a little exception to that rule with the Directional Swing solver.

The Directional Swing solver requires an extra parameter and overloads an optional one

```
/// <summary>
/// Solve the cain
/// </summary>
/// <param name="chain">The chain required</param>
/// <param name="lookAtAxis">Which axis of the end effector to consider</param>
public static void Process(Core.Chain chain, Vector3 lookAtAxis)
{
    Process(chain, lookAtAxis, chain.GetEndEffector());
}
```

```
/// <summary>
/// Solve the cain
/// </summary>
/// <param name="chain">The chain required</param>
/// <param name="lookAtAxis">Which axis of the end effector to consider</param>
/// <param name="offsetObj">Offset the end effector by this Transform (optional)</param>
public static void Process(Core.Chain chain, Vector3 lookAtAxis, Transform offsetObj)
{
    for (int i = 0; i < chain.iterations; i++)
    {
        Solve(chain, offsetObj, lookAtAxis);
    }
}
```

The first Method is the default: that requires a vector3 LookAt Axis argument, which will be the axis on the end effector we want to aim at. Keep in mind that this axis must be in world space so you would typically input something like : vector3.forward, vector3.right, vector3.left and so on

The second overload method: gives you an optional offset Transform object. What it does, is simply use this offsetObj as a virtual end effector.

Some use cases of the second overload method: is for example when you want to orient the torso of player so that a gun (which is in his hands) look at the target, but in same time, you don't want the gun to be included in the Chain because it does not belong there. You simply build a chain till the neck, most upper spine or whatever, call the solver and supply it with the weapon transform as its chain offset.