Unity - inverse kinematics for robot arms

- Inverse Kinematics Scripts
- Pick-and-place and IK Controller
- Customizing the IKController for Different Applications

Inverse Kinematics Scripts

Purpose

This set of scripts provides inverse kinematics solutions tailored for the UR10 robot arm, complemented by a custom actuator. It's designed to facilitate the visualization of the robot grasping and positioning boxes conveyed by a belt system.

Prerequisites

Unity Setup:

Ensure your Unity project is initialized using the 3D URP (Universal Render Pipeline) template. This script was developed and tested for Unity Editor version `2021.3.19f1` and as such, stability of this script in future versions is not guaranteed. The scripts can be downloaded, along with a demo project, can be downloaded from the DTLab GitLab page and then copied to the new Unity project. The essential scripts for Inverse Kinematics, without the pick-and-place functionality, are:

- `IKController` -> to control the robot arm's movement
- `DisableGravityForJoints` -> to override the physics built into the URDF model
- `MoveAlongPath` -> to move the box along the conveyor belt

Required Packages:

For effective robot arm simulation and interaction, integrate the following packages:

- ROS TCP Connector
- Unity Robotics Visualizations
- URDF Importer

All the above packages can be sourced and installed from Unity Robotics Hub on GitHub. Alternatively, they can be imported from locally cloned repositories housing the packages.

Robot Arm Integration:

- Using a URDF File: If your robot arm configuration is stored in a URDF file, navigate to the asset within Unity. Right-click and choose "Import Robot from Selected URDF file". This action will generate a 3D visual of the robot arm, incorporating the designated joints and their inherent physical constraints.
- Without a URDF File: If you possess only a 3D render of the robot, devoid of joint articulation, consider importing this model into Blender. Within Blender, introduce "bones" to your model. These bones will enable the various robot segments to exhibit mobility and flexibility.

Setup and Installation

1. **Script Integration:** Begin by adding the IK script to your Unity project.
2. **Setting up the IK Controller:**
   - Create a new empty GameObject within your scene.
   - Rename this GameObject to IK Controller.
   - Attach the IK Controller script to this GameObject.
3. **Configuration:**
   - Assign the appropriate GameObjects to the Actuator and IK Target fields.
   - For scenarios requiring multiple targets, ensure you populate the Target Array with the relevant GameObjects in the order they should be referenced.
4. **Joints Array Setup:**
   - Populate the Joints array with all individual joints from the robot arm.
   - Start with the base link (root of the arm) as Element 0 in the array.
   - Conclude the array with the actuator as the final element.

By following these steps, you will have successfully set up the IK Controller to interact with your robot arm in Unity.

Pick-and-place and IK Controller

Overview:

The `IKController` class provides an inverse kinematics solution for robotic arms in Unity. It allows the robotic arm to adjust its joints to reach or point to a specific target position. This is achieved through the gradient descent algorithm.

Properties:

- **_targetGameObject**: An array of GameObjects representing the targets the robotic arm should move towards.
- **actuator**: The GameObject representing the end effector of the robotic arm.
- **ikTarget**: The GameObject representing the inverse kinematics target.
- **endPoints**: An array of Transforms that indicate the end points of the robotic arm.
- **_targetTransform**: An array of Transforms associated with the `_targetGameObject`.
- **Joints**: An array representing the joints of the robotic arm. The joints determine the rotation axis.
- **Angles**: An array of floats representing the current angles of the robotic arm joints.
- **InverseKinematicController**: A GameObject that serves as the controller for inverse kinematics.
- **suctionCupTransform**: Transform component of the actuator.
- **delayedIKTarget**: An instance of `DelayedIKTargetUpdate` attached to the current GameObject.
- **collisionPenalty**: A penalty score added when joints come too close together to avoid self-collision.
- **target**: The target Vector3 position that the robotic arm should move towards.

Constants:

- **SamplingDistance**: The distance for sampling in the gradient descent algorithm.

- **LearningRate**: The rate at which the angles are adjusted during the gradient descent.
- **DistanceThreshold**: The threshold under which the arm is considered to have reached the target in terms of distance.
- **AngleThreshold**: The threshold under which the arm is considered to have reached the target in terms of angle.

Methods:

`Start()`

Initializes the `_targetTransform` array by fetching the `Transform` component of each target GameObject. It also initializes the `suctionCupTransform` and sets the initial target for the robotic arm based on the `Dropped` flag.

`StartMovementWithDelay()`

Starts the inverse kinematics movement after a delay. Not used in the demo implementation, but can be used as an alternative to th distance-based default.

`InverseKinematicCoroutine()`

A coroutine that updates the target position to the position of the `ikTarget` and adjusts the angles of the joints to move towards this new target.

`Update()`

Called once per frame. Adjusts the target position based on various conditions and executes the inverse kinematics algorithm.

`ForwardKinematics(float[] angles)`

Given an array of joint angles, it computes the position of the end effector using forward kinematics.

`DistanceFromTarget(Vector3 target, float[] angles)`

Calculates the distance between the target position and the end effector's position.

`AngleWithTarget(Vector3 target, float[] angles)`

Calculates the angle between the target position and the end effector's position.

`ErrorFunction(GameObject target, float[] angles)`

Calculates the error between the end effector's position and rotation and the target's position and rotation.

`PartialGradient(Vector3 target, float[] angles, int i)`

Computes the partial gradient of the error with respect to a specific joint angle.

`InverseKinematics(Vector3 target, float[] angles)`

The core method for the inverse kinematics solution. Adjusts the joint angles so that the end effector moves closer to the target position.

Remarks:

The script adjusts the robotic arm's angles using the gradient descent algorithm to minimize the error function. The error function encapsulates the distance between the end effector and the target and potential penalties due to near-collisions between joints.

Ensure to have the required targets and joint GameObjects correctly set in the Unity Editor for this script to work as intended.

Customizing the IKController for Different Applications

The `IKController` script is a versatile framework for inverse kinematics (IK) and can be customized for various applications. Depending on your needs, you might want to adjust several properties or change the optimization function. Here’s how you can do so:

Serialized Fields for Editor Exposure:

Unity's Inspector is a powerful tool to tweak properties in real-time. By making variables into `[SerializeField]`, even if they are private, you can expose them in the Unity Editor. This makes it easier to adjust values during runtime and tune your IK without having to modify the script directly.

For example, if you want to expose the `LearningRate` variable, simply modify its declaration:

```
[SerializeField] private const float LearningRate = 50f;
```

Now, `LearningRate` will appear in the Unity Inspector, allowing you to modify its value through the Unity Editor interface.

Tweaking Joint Properties:

Each joint (`ArmJoint`) is a critical component in the IK system. You might need to adjust the following properties for different robots or characters:

- **Rotation Axis**: Determines which axis (x, y, z) the joint will rotate around.
- **Min/Max Angle**: Constraints on how much the joint can rotate. This is essential to ensure that the joint doesn't rotate beyond its physical or desired limits.
- **StartOffset**: The starting position offset of the joint relative to its parent.

You can add these as serialized fields if you want to tweak them in the Unity Editor directly.

Enhancing the Error Function:

The current error function in the `IKController` script primarily focuses on the distance and angle to the target. However, in many real-world scenarios, other factors might be crucial.

Collision Handling:

Collisions between robot joints or between a robot and external objects can be problematic. You can enhance the error function to penalize configurations where joints overlap or come too close to other objects. To do this, integrate a collision checker that returns a high penalty value if a collision is detected.

For example:

```
if (JointCollisionDetected(i))
{
    penalty += HighCollisionPenaltyValue;
}
```

Energy Consumption or Efficiency:

For robots where energy efficiency is crucial, you might want to add a term that penalizes configurations requiring more energy. This could be based on joint angles, rotation speed, or other robot-specific parameters.

4. Changing Optimization Techniques:

The provided script uses gradient descent, which might not be optimal for all scenarios. Depending on your robot's complexity and the required precision, you might want to explore other optimization techniques or even machine learning-based approaches to find the best configuration.

Introducing New Constraints:

Depending on the robot's design, there might be specific postures or configurations that are undesirable or impossible. By adding custom constraints to the optimization process, you can ensure that the IK solution respects these constraints.

Remember, the `IKController` is a starting point. Depending on the complexity of your application, robot design, and requirements, you might need to expand and customize it further. Always test your changes in a controlled environment to ensure that the robot behaves as expected.