Performance vs Quality and some advices

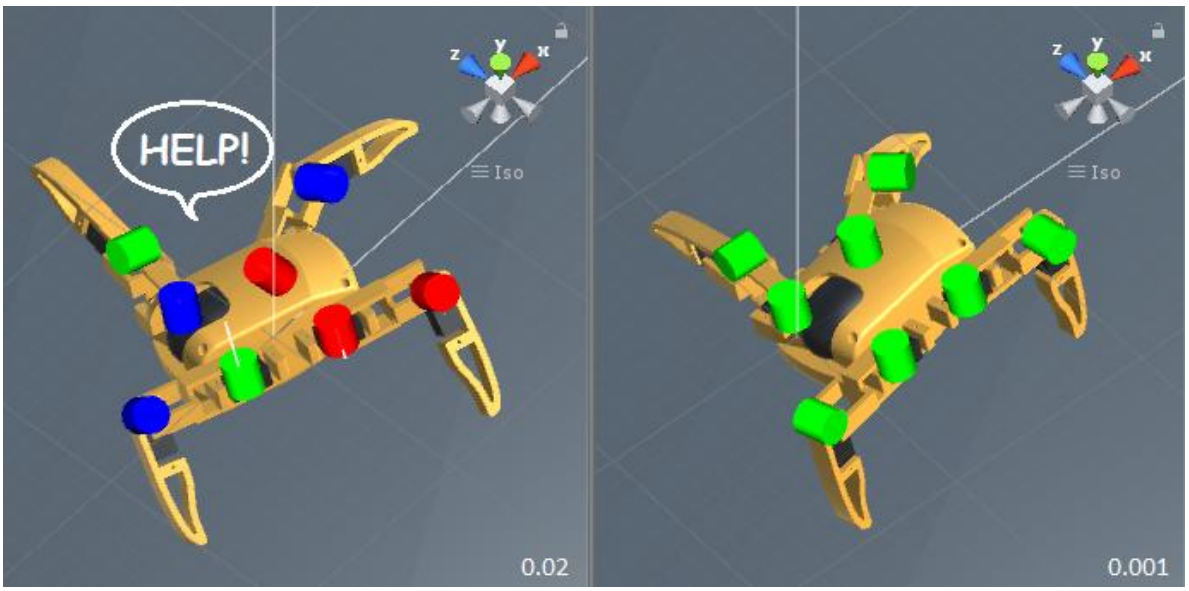
The main setting to control the quality of physics simulations in Unity is the Fixed Timestep (Edit → Project Settings → Time). It is the timestep in seconds used for all physics calculations including collision detection and calling FixedUpdate() on every component in game, therefore it affects performance dramatically. For example, if Fixed Timestep equals to 0.001 then each servo motor will need to update its velocity one thousand times per second. The default value of the FixedTimestep is 0.02 and it's perfect for most games developed in Unity, but if you try to run the simulation with this settings, the joints will jitter and break and the servos won't be able do track target angles. This is because we require a precise control over joints and rigidbodies and need to update physics much more frequent.

This why Fixed Timestep of about 0.001 should be used.

Now lets talk about performance. Timestep of 0.001 allows the simulation of a single quadruped robot run smoothly even on the old machines, but as you create more copies of the robot, you will notice that the simulation start freezing. On my Intel i5 3500GHz I was able to simulate 20 quadruped robots with Fixed Timestep equals to 0.002. Carry your own experiments to find your best settings, but I wouldn't recommend increasing Fixed Timestep above 0.003, as it might lead to a poor simulation quality.

Changing Fixed Timestep might lead to joint jittering and require tweaking some other settings, especially Velocity PID and Friction Profile. This is because time period is an important parameter for all discreet calculations.

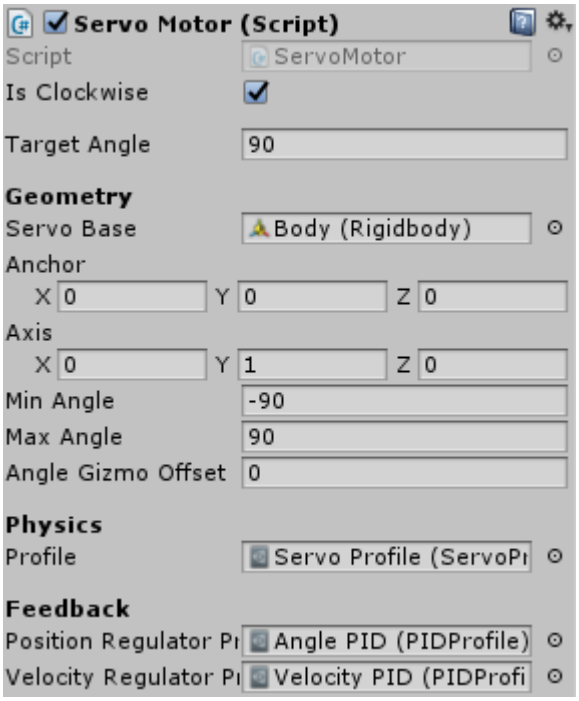
You should always use realistic scale for all dimensions and physical quantities. 1 size unit equals to 1 meter, forces are set in newtons, torques in newtons per meter, time in seconds, mass in kilograms. Another important thing you should know about is that PhysX joints tend to jitter if they connect bodies with large difference in masses. If the difference is 10 times or more, you will get a poor simulation quality.



Fixed Timestep 0.02s vs 0.001s. Notice that robot on the left has broken a couple of joints

Servo Motor

To create a servo motor, add ServoMotor component to a Rigidbody. On the simulation start it will create, automatically configure and control a HingeJoint component. You should not modify or control HingeJoint directly.



Before running the simulation you need to configure all of the following settings:
isClockwise – selects the positive direction of servo rotation.
servoBase – a Rigidbody this servo is rotating relative to. If none, servo will rotate relative to the world.
anchor – a centre of rotation set in local coordinates. It is visualized by the green cylinder.
axis – a direction of the axis of rotation set in local coordinates. If zero, X axis will be used by default. Axis is visualized by rotation of the green cylinder.
min and **maxAngle** - limits of rotation in degrees relative to initial position. Visualized with the blue arc.
angleGizmoOffset – does not affect servo operation, but is useful for visualization. Use this setting to rotate the blue arc to a position where it feels the most natural.
profile - ServoProfile is an asset used to configure common servo parameters. You can double click on this field to go to the selected asset. All changes made to this asset will be immediately applied to all servos currently using it.
positionRegulatorProfile – an asset used to tune the position PID controller. It is required for operation of the servo motor. The output of this controller is set as the target velocity of the joint, unless we have a...
velocityRegulatorProfile – which is an asset used to tune the velocity PID controller. If this field is set to None, then velocity PID is turned off... But why do we need a velocity PID? Unfortunately PhysX joints are quite bad at tracking the target velocity. It might work fine in some conditions, but if you apply an external load to the joint, It will stop even though it is not applying it's maximum torque yet. Even if the joint doesn't stop, you might notice that it is moving slower than it should. To fix this problem I used a velocity PID regulator, which basically implements the torque control. It is a quite sensitive regulator and it might lead to a small tremor (on the other side the real servos do have tremor too). Anyway it's up to you whether to use this regulator or not. To disable it, just select None in the profile field.
Both PositionRegulatorProfile and VelocityRegulator profile are assets and all of the changes made to them will be immediately applied to all servos using them.
targetAngle – the angle the servo is currently trying to reach. It's actually a private field, so you won't be able to control it from other scripts. Instead you should use SetAngle(float) method. This way the servo delay will be accounted.

Public Properties

float Range

The full range of servo rotation, calculated as maxAngle – minAngle

bool IsFixed

When fixed, servo configures HingeJoint so it doesn't rotate and acts like a FixedJoint component. False by default.

bool IsMotorEnabled

Disabling motor is similar to removing power from the servo. In this mode servo acts like a regular HingeJoint. True by default.

Public Methods

float GetServoVelocity()

Calculates servo velocity in degrees per second based on the actual Rigidbody velocity. Works similar to HingeJoint.velocity, but works properly in complex joint systems, in contrast to HingeJoint.velocity which breaks.

float GetServoAngle()

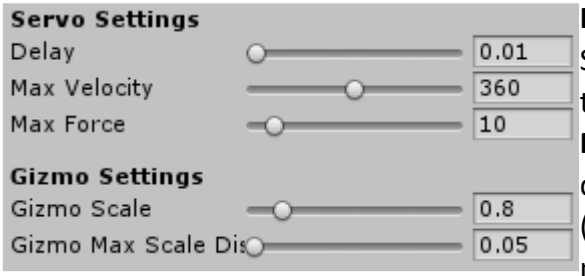
Calculates servo angle in degrees based on the actual Transform rotation. Works similar to HingeJoint.angle, but works properly in complex joint systems.

void SetAngle(float value)

Sets target rotation angle in degrees. Changes are applied after delay period.

ServoProfile

ServoProfile is an asset used to configure common servo parameters such as physics settings and visual appearance. You can store several assets for different servo configurations in your project. To create a new asset, right click on the Project window and select Create → ServoProfile. All changes made to this asset will be immediately applied to all servos currently using it. Changes made at Runtime will be preserved.



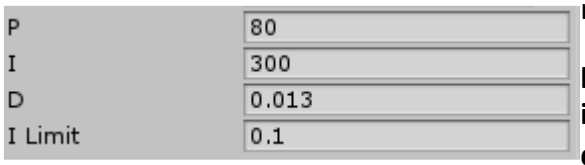
Delay – time in seconds between calling SetAngle(float) and actually applying the new target.

MaxVelocity – maximum servo velocity in degrees per second. If small values are used (less than 200 degrees/second), you might notice, that Unity fails in keeping the correct velocity. Either don't use small values or set an extremely low TimeStep, which helps, but takes CPU performance.

MaxForce – maximum servo torque in newtons per meter.
GizmoScale and **GizmoMaxScaleDistance** – affect the appearance of the debug visuals. The first one controls scale, and the second one controls the maximum distance after which they start getting smaller. Might work differently for different screens, so you might want to tune these after transferring your project to another computer.

PIDProfile

PIDProfile is an asset used to configure PID regulators. You can store several assets for different PID configurations in your project. To create a new asset, right click on the Project window and select Create → PIDProfile. All changes made to this asset will be immediately applied to all servos PIDs using it. Changes made at Runtime will be preserved.



p – proportional gain.
i – integral gain.
d – differential gain.
iLimit – integrator saturation limit.

PIDRegulator

While the previous class is only used for storing PID configuration, this one actually does all of the calculations. PID regulators are used by servos to track target position, but the class itself doesn't depend on ServoMotor, so if you need, you can use it for other purposes. If you prefer using the ServoMotor as a “black box”, you won't need to communicate with this class directly. All of the work is done inside of the ServoMotor.FixedUpdate().

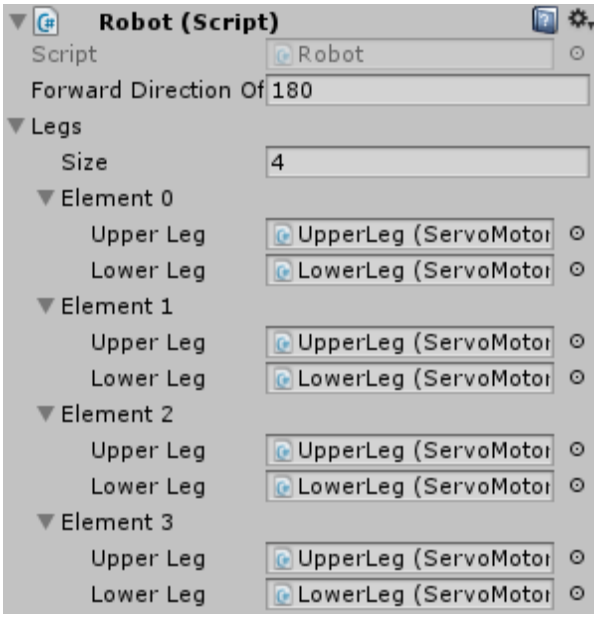
Public Methods

PIDRegulator (PIDProfile profile)
Constructor. Pass PID configuration to be used.

float Run (float target, float current, float deltaTime)
Runs a PID regulator and returns a correction value. Target is the value to be tracked, current is the current value, deltaTime is the time period in seconds since the last call.

Robot

Each robot should have one Robot component. The Transform of the GameObject with the Robot component will be used in distance and heading calculations, so make sure, this object actually moves with the robot body. For example in the demo scene the Robot component is attached to the Body GameObject, not the one called Robot. That's because, the Robot GameObject is just a parent for all of the robot parts. It's like a folder, which is convenient for keeping the project organized and dragging the robot around the scene in the editor, but the Robot GameObject doesn't actually move with the robot in runtime, when the physics is applied, as it doesn't have a Rigidbody component. The Body on the other side, represents the actual body of the robot. It's center is located in the center of the robot, and it's angle around the vertical axis is the robot heading. That's why we use Body GameObject for the Robot component.



forwardDirectionOffset – choose the direction in which the walked distance should be calculated. Visualized as the white arrow.
legs – an array of robot legs. Each of them has two properties:
upperLeg – the ServoMotor of the upper leg.
lowerLeg – the ServoMotor of the lower leg.
Robot component has a few convenient methods for control of the legs sections, but this array is how you access robot legs most of the time.
For example, if you want to turn the lower leg of the leg 0 to a new position, you go: robot.legs[0].lowerLeg.SetAngle(float value).

Public Methods

float GetHeading()

Returns current heading in degrees relative to initial rotation of the robot.

float GetDistance()

Returns the distance walked along the forward direction in meters. Visualized as the black line from the robot initial position.

void FixLegs(), void FixUpperLegs(), void FixLowerLegs()

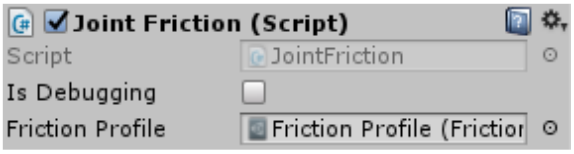
Fixes legs servos.

void UnfixLegs(), void UnfixUpperLegs(), void UnfixLowerLegs()

Unfixes legs servos.

JointFriction

Adds viscous and Coulomb friction to the servo. Requires a ServoMotor component on the same GameObject.

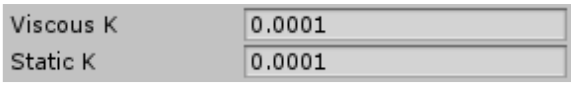


isDebugging – when checked, the component draws white rays visualizing the direction and the amount of the applied friction torque.

frictionProfile – an asset with the current friction coefficients.

FrictionProfile

FrictionProfile is an asset used to configure joint friction. You can store several assets for different friction settings in your project. To create a new asset, right click on the Project window and select Create → FrictionProfile. All changes made to this asset will be immediately applied to all joints using it. Changes made at Runtime will be preserved.



viscousK – viscous gain [n / m] * [sec / deg]
staticK – static friction gain [n / m]

Both values should be kept low, otherwise unexpected behavior might occur.