KINESIS

Table of Contents

Basics

Getting Started

Installation and Setup

1. Create a new Unity project or open an existing one.
2. Download and import the Kinesis package into your project. That's it.

Beginner Orientation

The recommended way to start is to review the rest of the Basics and Fundamentals sections for a high-level overview of the package and its architecture. After that, check out the Tutorial and the Demo scenes included with the package, and read the Tip & Tricks section when you're ready to start experimenting.

If you're reading a local version of this documentation, you can find the latest version online at https://squeakyspacebar.github.io/kinesis-doc. You can also find the Doxygen-generated API documentation in the package directory under `Kinesis/Documentation/html/index.html` or online at https://squeakyspacebar.github.io/kinesis-doc/api/.

Package Overview

Package Contents

- `Demo`: Additional assets for the tutorial and demo scenes.
  - `Character`: Character model assets.
  - `Editor`: Custom editor and property drawer scripts for demo components.
  - `Environment`: Environment assets.
  - `Prefabs`: Prefabs for the Tutorial and Demo scenes.
  - `Scenes`: The Tutorial and Demo scenes.
  - `Scripts`: Demo scripts and components.
- `Documentation`: Locally packaged copy of documentation.
- `Editor`: Custom editor and property drawer scripts to improve the basic editor experience.
- `Scripts`: The core muscle model implementation, additional components, and supporting code.
  - `Attributes`: Basic attributes, mainly for controlling the display of fields in the Editor.
  - `Components`: Non-core components.
  - `Core`: Core scripts that implement the muscle model.
  - `Exceptions`: Custom exceptions thrown by Kinesis.
  - `Library`: Generic reusable code.

Code Overview

Namespaces

All of the code within Kinesis is namespaced.

Namespace	Description
`Kinesis`	Root namespace.
`Kinesis.Core`	Namespace for core components.
`Kinesis.Components`	Namespace for non-core components.
`Kinesis.Demo`	Namespace for everything related to the demo.
`Kinesis.UnityInterface`	Namespace for custom editors and property drawers.

Components

Core

- **Muscle Tendon Unit**: This component is the core of the muscle model implementation and is how you give an object muscle behavior.

Non-Core

- **Joint Data**: Acts as a data container for conveniently referencing data related to muscle manipulation. As part of that responsibility, it scans and caches a list of joint-spanning muscle segments for quick lookup. Requires a **Joint** component on any GameObject it is attached to.
- **Muscle Group**: This is a (currently) minimal component meant as a convenient way to mark a GameObject as a container for organizing/grouping muscles.
- **Muscle Render Gizmo**: This component visualizes muscle-related data in the Scene view such as muscle node positions, muscle segment lines, lever arms, and joint torques, with several options to customize the display of information. The component also registers a callbackin the Scene view—triggered by hierarchy changes—that scans all muscle objects to keep an updated list.
- **Muscle Rig**: Meant to be attached to a skeleton's root GameObject. It scans child GameObjects and attaches a **Joint Data** component if a corresponding **Joint** component is detected. The component performs the inverse when removed and removes any **Joint Data** components attached to child GameObjects.
- **Muscle Stimulator**: This component provides the ability to interact directly with muscles from the Unity interface, and can be used to manually set excitation and run physics simulation for specific muscles. Requires a **Muscle Tendon Unit** component on any GameObject it is attached to.

Exceptions

- **MuscleComponentLengthException**: Thrown when a muscle element's length value exceeds the total muscle length. This can happen when the physics are moving too quickly and the muscle gets into a bad state, but should not occur often under normal operation.
- **MuscleNodeMissingBoneException**: Thrown when attempting to reference the bone object of a muscle node when it has not been set (i.e. is `null`).

Fundamentals

Model Architecture

This section primarily aims to provide a high-level introduction to the muscle model behind Kinesis. For a more detailed look, please consult the cited literature and source code.

Introduction to Hill-Type Muscle Models

The term "Hill-type muscle model" or "Hill muscle model" refers to a certain kind of muscle dynamics model which is typically composed of a system of contractile, parallel, and serial components. The exact component configurations can vary depending on the model.

The history of Hill-type models stretches back to the early 20th century and a paper by physiologist A.V. Hill [(Hill, 1938)](/

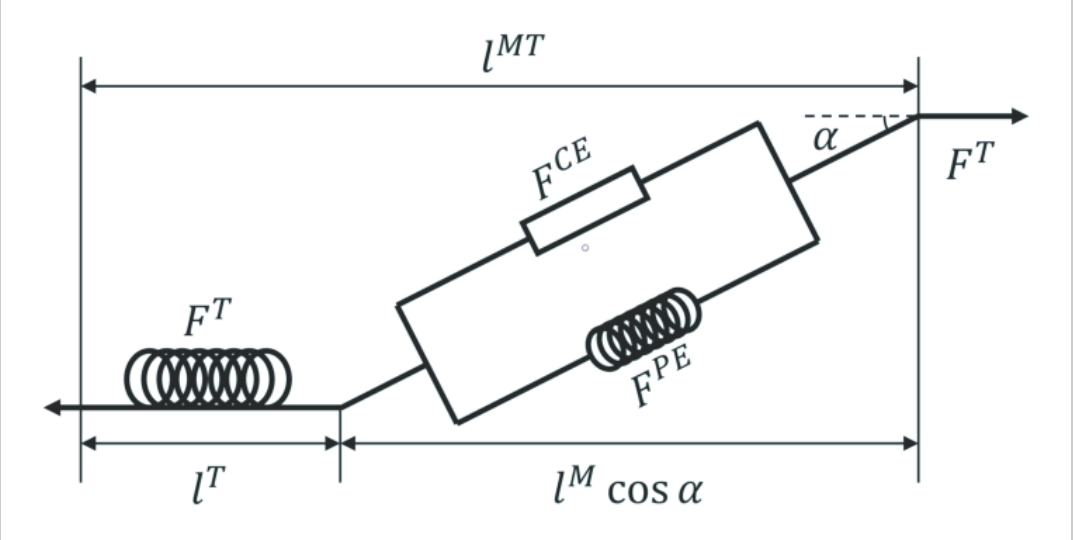


Figure 1 - Structure of a Hill-Type Muscle Model by (Zhou, et al., 2019) is licensed under CC BY 4.0

We'll start by looking at the three-element Hill-type model in Kinesis. This model typically consists of a contractile element (CE) in combination with two nonlinear spring elements: the parallel element (PE), as it is parallel to the CE, and the serial element (SE), as it is in series to both the CE and PE together. Hill-type models often focus on the muscle contraction force, which depends on the muscle's physical state at a given moment (i.e. CE length and contraction velocity).

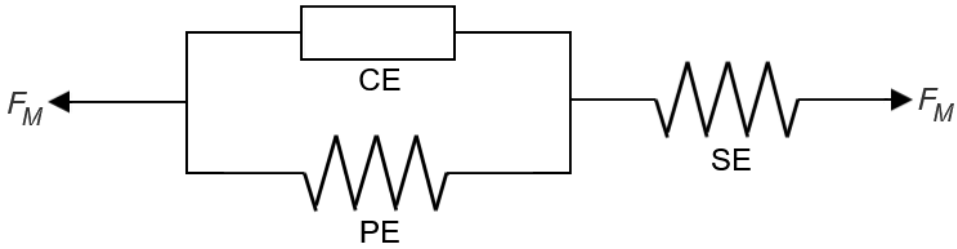


Figure 2 - Three-Element Hill Model in Kinesis

Relating each element to a biological analogue may be helpful:

- The CE represents muscle fibers that contract when the muscle is activated.
- The PE represents passive connective tissue around the contractile muscle fibers that are elastic and resist stretching or compression. Being passive means that the element does not respond directly to activation.
- The SE represents the tendon (also passive and elastic) that attaches muscle to bone.

A caveat though that these biological interpretations are merely for mental convenience. There are of course much more complex and detailed models of muscle structure and function depending on the characteristics being investigated.

Also, a note if you're unfamiliar, PE and SE elements are often seen only consisting of a parallel *elastic* element (PEE) and a serial *elastic* element (SEE) respectively, but it's not uncommon to encounter models that also feature a parallel *damping* element (PDE) and/or a serial *damping* element (SDE); both the elastic and damping elements are typically considered component parts of their broader respective elements (i.e. the PE consists of the PEE *and* PDE, and the SE consists of the SEE *and* SDE).

Overview of the Multi-Segment Hill-Type Muscle Model in Kinesis

The muscle model in Kinesis is heavily based on the descriptions in [(Geijtenbeek, et al., 2013)] (/

Each muscle node has a position defined relative to a body part or "bone" and maintains that relative position to the bone as it moves. Each segment is defined by exactly a pair of nodes, and those with their nodes on separate (but adjacent) bodies/bones are said to *span* the joint that connects the bodies/bones. These spanning segments are important for calculating the

joint torques generated by the muscle.

Each muscle receives an *excitation signal*, which is then transformed by *activation dynamics* into a *muscle activation*, which is what drives muscle contraction. Both the CE length and contraction velocity are used to determine the contractile force being generated, which is then used to calculate the joint torques generated by the muscle.

The joint torques are calculated based on the following. Each muscle's contractile force (which is a scalar magnitude value) is applied to the *line of action*, which is a unit vector pointing from the spanning muscle segment's tail node (the one with the higher index) toward its head node. Put another way, the magnitude of the muscle's contractile force scales the line of action unit vector. The *lever arm* is simply the vector from the joint toward the spanning muscle segment's tail node.

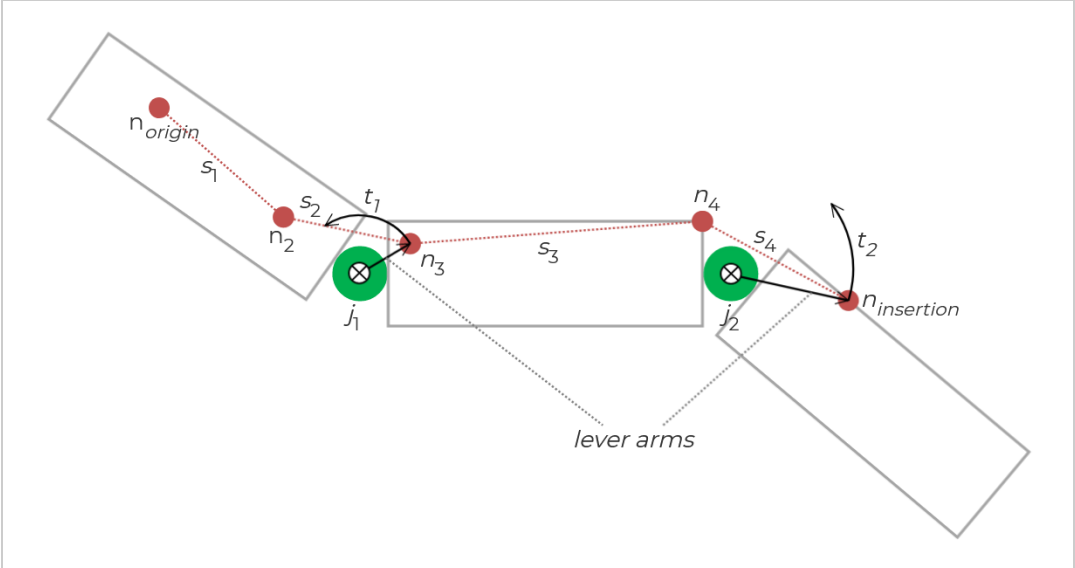


Figure 3 - Multi-segment muscle model

Core

This section aims to provide a basic overview of the core muscle model implementation. Please refer to the API documentation (either the version included with the project or online) and the source code for further details.

Concepts

Please refer to the previous Overview of the Multi-Segment Hill-Type Muscle Model in Kinesis section for a conceptual brief.

Directory and File Structure

In the `Kinesis/Scripts/Core` directory, you should find the scripts that comprise the core implementation of the muscle model:

- `MuscleNode.cs`: Contains the `MuscleNode` class representing muscle nodes.

- `MuscleSegment.cs`: Contains the `MuscleSegment` class representing muscle segments.
- `MuscleTendonUnit.cs`: Contains the `MuscleTendonUnit` Monobehaviour class implementing the actual muscle model.

Code Structure

Muscle Node

Muscle nodes are represented by the `MuscleNode` class. It has a very basic structure: - `bone`: Represents the "bone" that the muscle node is "attached" to. - `offset`: The relative position (in the bone's local coordinate space) that the muscle node maintains.

Muscle Segment

Muscle segments are represented by the `MuscleSegment` class. - `prevSegment`: References the adjacent muscle segment toward the muscle origin along the muscle path. - `parentMuscle`: References the `MuscleTendonUnit` the muscle segment is a part of. - `head`: References the `MuscleNode` toward the muscle origin along the muscle path. - `tail`: References the `MuscleNode` toward the muscle insertion along the muscle path. - `joint`: If the muscle segment spans a joint, this references the **Joint** spanned by the muscle segment. - `jointAnchorBody`: References the **Rigidbody** that is the anchor for `joint`.

Muscle Tendon Unit

The `MuscleTendonUnit` class represents a MTU (muscle-tendon unit, which is the "muscle" we refere to in this documentation) as a whole. It is a fairly large technical class and would take a lot more content to explain in-depth, so here are some of the more important highlights:

- The public `muscleNodes` and `muscleSegments` fields contain the data representing the muscle path. > **Caveat:** Currently, a **Muscle Segment** should not be defined such that it would span more than a single joint.

  > **Note:** The `MuscleTendonUnitEditor` script listens for muscle node updates and should automatically trigger a rebuild of the muscle segment list.

- The `CalculateJointTorques()` and `ApplyJointTorques()` methods are used to iterate the muscle simulation. The functionality was separated between calculation and application to allow for more control and extensibility. Please check out the **Muscle Simulation** component in the included [Demo Scene](

- The `CalculateJointTorques()` method takes an excitation signal and a dictionary to store the calculated results. The dictionary uses spanning muscle segments as keys and the joint torques that should be applied as the values.

- The `ApplyJointTorques()` *static* method takes a dictionary in the same form as `CalculateJointTorques()`. It parses the given dictionary to apply the joint torques to the proper rigid bodies. It is a static method as it does not require any instanced data and receives everything it needs through its arguments.

- There are a number of constants defined in that class. Most of these are part of the more opaque technical details of the muscle model and are not meant to be adjusted casually.
- `normalizedSEESlackLength` controls the initial ratio of SEE to CE length out of the total muscle length.
- `maxIsometricForce` most directly adjusts the force a muscle is able to generate under contraction.
- `activationConstant` affects the activation rate in the activation dynamics.

Tutorial

Loading the Tutorial Scene

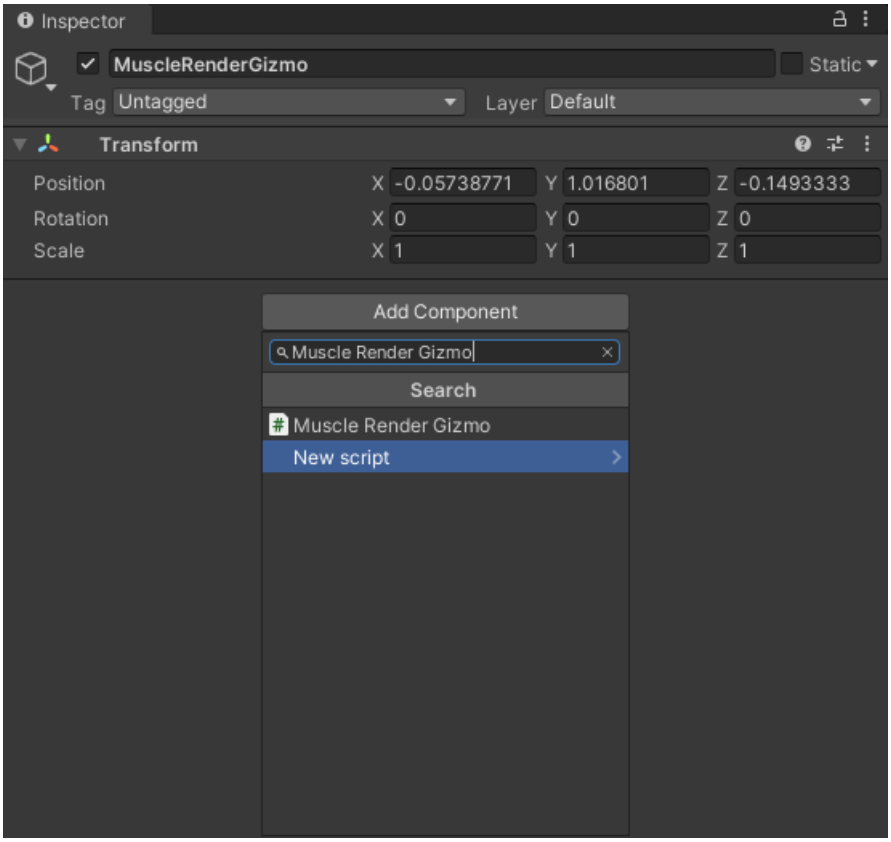
First, create a new project for the tutorial to load the tutorial scene. Kinesis comes with a simple scene for the purposes of this tutorial under the path `Kinesis/Demo/Scenes/TutorialScene.unity`, which you can open by either navigating to it via **File > Open Scene** or the **Project** window.

Setting Up Muscle Visualization

Kinesis comes with a **Muscle Render Gizmo** component that adds muscle visualization elements to the Scene view.

1. Create a new empty `GameObject` and rename it to "MuscleRenderGizmo" (the name isn't critical).
2. Add a **Muscle Render Gizmo** component to the new object.

- Select the "MuscleRenderGizmo" object to focus it in the **Inspector** window.
- Attach the **Muscle Render Gizmo** component. Select the **Add Component** button in the **Inspector** window, then type "Muscle Render Gizmo" into the search field when it appears. The **Muscle Render Gizmo** component should appear as the top search result. Select the component in the search result to add it to the object. We'll leave the default settings for the tutorial.



3. Your project hierarchy should now look something like this:

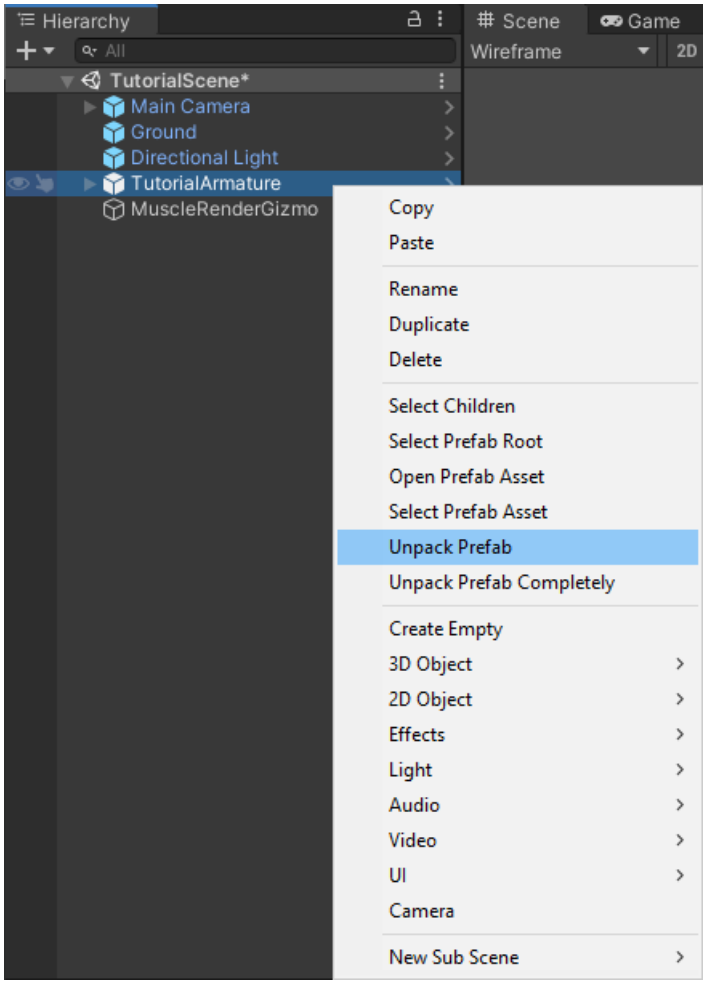
Setting Up a Basic Muscle

For the tutorial, we're using a slightly modified character model from Unity's Starter Assets - Third Person Character Controller, but the following steps should work for pretty much any other model with a typical humanoid skeleton.

Unpacking the Prefab

In order to freely edit the tutorial model, first we'll need to unpack the prefab to unlink it.

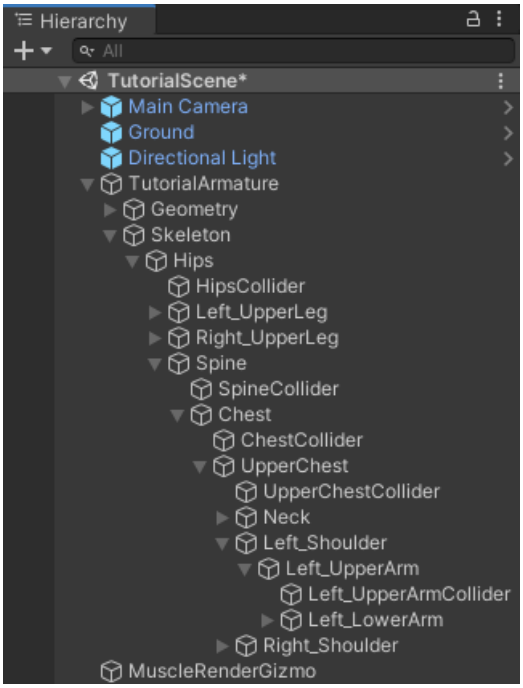
1. Find the "TutorialArmature" object in the Hierarchy window and secondary click on it to open up a context menu.
2. Select the *Unpack Prefab* option.



Creating a Jointed Limb

Before you can create a functioning muscle, you'll first need at least two **Rigidbody** objects connected by a **Joint**. We'll set up the left arm's upper and lower sections with **Rigidbody** components and connect them via a **CharacterJoint** component that represents the elbow.

1. In the **Hierarchy** window, expand out the object hierarchy under "Skeleton" until both the "Left_UpperArm" and "Left_LowerArm" objects are exposed. The path should be "TutorialArmature" > "Skeleton" > "Hips" > "Spine" > "Chest" > "UpperChest" > "Left_Shoulder" > "Left_UpperArm" > "Left_LowerArm".



2. Add a **Rigidbody** component to the "Left_UpperArm" object.

- Select the "Left_UpperArm" object to focus it in the **Inspector** window.
- Select the **Add Component** button in the **Inspector** window, then type "Rigidbody" into the search field when it appears. The **Rigidbody** component should appear as the top search result. Select the component in the search result to add it to the "Left_UpperArm" object.

3. Configure the **Rigidbody** component on the "Left_UpperArm" object:

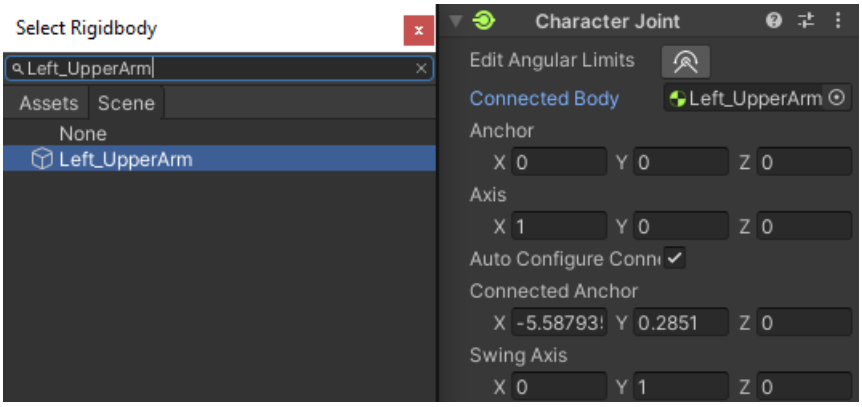
- Set *Is Kinematic* to `true`. The reason for this is that the rigidbody's movement is not currently limited by a connected joint or any other constraint, so it'll move independently of the rest of the model if we don't constrain it. For this tutorial, we are only testing the muscle's effect on the lower arm and we'll want to "pin" the upper arm in space.

4. Add both a **Rigidbody** and **Character Joint** component to the "Left_LowerArm" object.
5. Configure the **Rigidbody** component on the "Left_LowerArm" object:

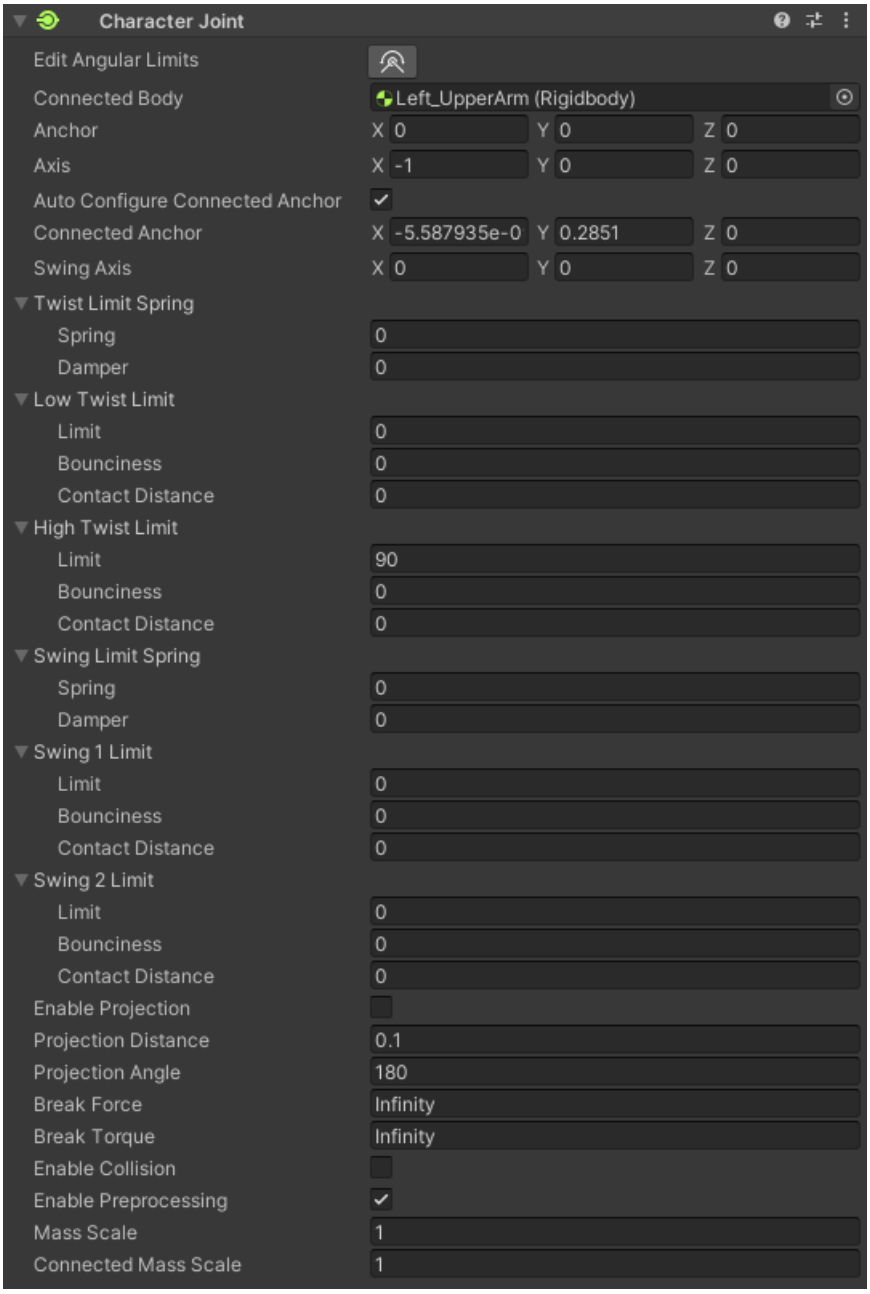
- Set both *Drag* and *Angular Drag* to 10. We really should be adjusting the inertia tensor, but that's outside the scope of this tutorial.

6. Configure the **Character Joint** component on the "Left_LowerArm" object:

- Set the *Connected Body* field to reference the "Left_UpperArm" object.
  - With the "Left_LowerArm" object in focus in the **Inspector** window, navigate to the *Connected Body* field. You can either drag the "Left_UpperArm" object from the **Hierarchy** window and drop it onto the *Connected Body* field, or you can select the circular bullseye icon to open the **Object Picker** and search for the "Left_UpperArm" object (if so, make sure to have the Scene tab selected to display the results we want).



- Configure the **Character Joint** constraints. We won't get too involved with configuring motion constraints, but for this tutorial, we'll at least want to:
  - Set *Axis* to X: -1, Y: 0, Z: 0.
  - Set *Swing Axis* to X: 0, Y: 0, Z: 0.
  - Set *Limit* under *Low Twist Limit* to 0 and under *High Twist Limit* to 90.
  - Set *Limit* under both *Swing Limit 1* and *Swing Limit 2* to 0. > **Note:** If you're not using the tutorial model, please be aware the exact axes and constraint angles you want may be different.



Creating a Muscle

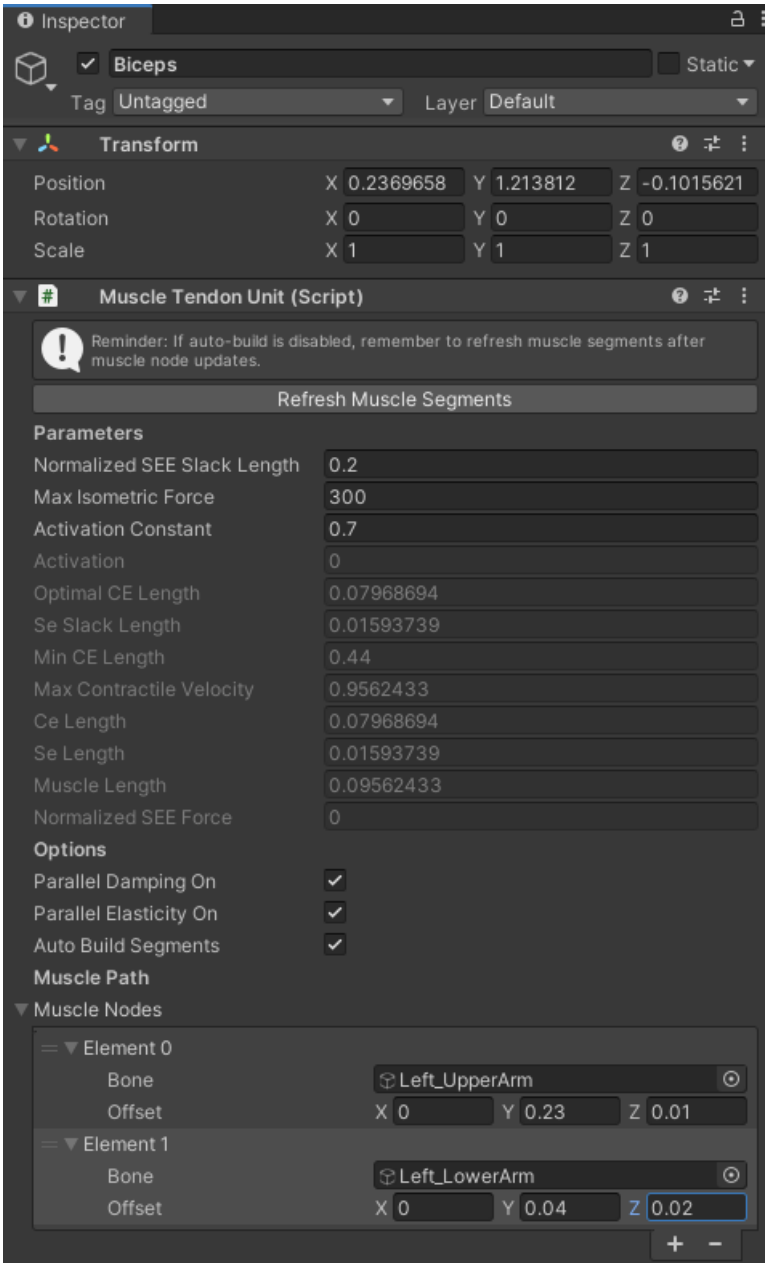
1. Create a new empty `GameObject`, rename it to “Biceps”, and attach the **Muscle Tendon Unit** component to it. You can either use the **Add Component** picker and search for “Muscle Tendon Unit,” or you can drag the script at `Kinesis/Scripts/Core/MuscleTendonUnit` from the **Project** window and drop it over the **Inspector** window.
2. Once the **Muscle Tendon Unit** component as been added to your object, you’ll need to define some muscle nodes to define the muscle path. In the **Inspector Window**, go to

the *Muscle Nodes* field and select the plus icon to add a new muscle node item to the list.
> **Note:** Muscle nodes are expected to be defined in order from origin to insertion.

3. Configure the first muscle node:

- Set the *Bone* field of the first muscle node to reference the “Left_UpperArm” object. In Scene view, the **Muscle Render Gizmo** should now be displaying a spherical indicator highlighting the muscle node’s position (this should initially be at the local origin at the shoulder socket).
- Set *Offset* to X: 0, Y: 0.23, Z: 0.01.

4. Add a second muscle node.
5. Configure the second muscle node:
6. Set the *Bone* field of the second muscle node to reference the “Left_LowerArm” object.
7. Set *Offset* to X:0, Y: 0.04, Z: 0.02.
8. Your Inspector window should now look something like this:

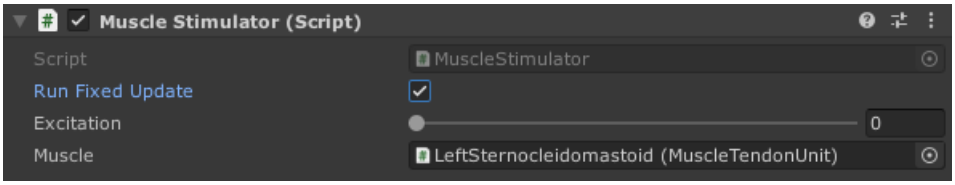


Testing Muscle Movement

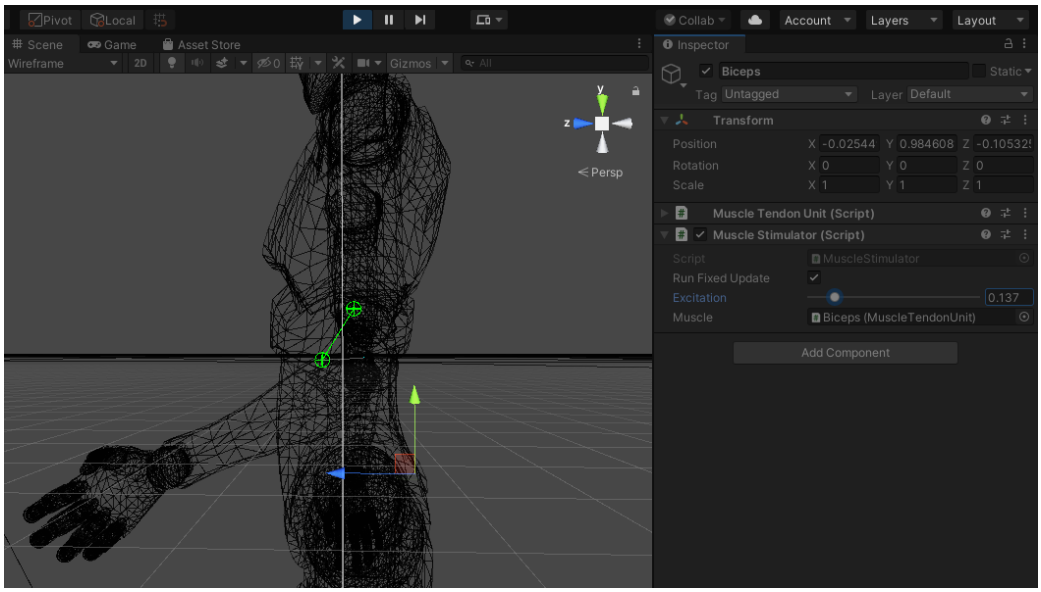
Once you've defined your muscle path, it's time to test its movement. Kinesis comes with a **Muscle Stimulator** component for convenience. It allows you to directly run physics simulation and set excitation for individual muscles from the **Editor**.

1. Add a **Muscle Stimulator** component to the "Biceps" muscle object. > **Tip:** It might help at this point to collapse the **Muscle Tendon Unit** component in the Inspector window by selecting the triangular icon next to the name in the header bar. This will help declutter your view.
2. Configure the **Muscle Stimulator** component:

- Set *Run Fixed Update* to `true`.



3. Enter Play Mode and bring the left arm into view in the Scene view.
4. With the "Biceps" muscle object in focus in the **Inspector** window, navigate to the **Muscle Stimulator** component and change the value of the *Excitation* field by adjusting the slider. You should see the lower arm react to the changes in excitation.



That's the end of the tutorial. Feel free to continue to experiment with the tutorial environment or move on to the Demo Scene.

Demo Scene

Demo Overview

Kinesis comes with a simple demonstration scene that includes a model rigged with several muscles. Please refer to the Package Contents section for the demo directory layout. > **Caveat:** The model in the demo is not meant to be taken as anatomically precise or accurate. The model is only generally humanoid, the skeleton is fairly simple, and the muscle paths are "looks good enough" estimates adapted to the morphology of the model.

> The parameters of the muscles themselves are also currently not set against any particular reference, so their responses may not be quite true to life. Also, not all of the muscles were able to receive equal amounts of testing attention, so there are certain to be configurations that don't behave well. Feel free to experiment away, just be aware that the model is merely a rough demo.

The included "DemoArmature" has two important children in its hierarchy:

- "Skeleton": Contains the skeleton hierarchy. It has been configured with colliders, **Rigidbody** components, and **CharacterJoint** components for the demo.

  > **Caveat:** Since there is no automated muscle controller out-of-the-box, unless you have a controller of our own, you'll have to constrain parts of the model if you want it to "stand upright." I typically set *Is Kinematic* to `true` on the **Rigidbody** components of the "Hips," "Spine," "Chest," and "Upper Chest" in order to "pin" them in place when working with limbs in isolation.

- "Musculature": Contains the muscle hierarchy. You'll find all of the muscles here, organized under muscle groups.

Demo Components

- `Camera Controller`: Provides a simple free-roam camera controller for navigating the Game view in Play Mode.
- `Muscle Simulation`: Provides a centralized loop that runs physics calculations for all of the muscles in the scene and feeds in any excitations from the **Muscle Stimulator** components connected to the muscles.

Loading the Demo Scene

You'll find the Demo Scene under `Kinesis/Demo/Scenes/DemoScene.unity`, which you can open by either navigating to it via **File > Open Scene** or the **Project** window.

You can either create a new project or use an existing one to open the scene.

Viewing the Demo Scene

The demo scene has a **Muscle Render Gizmo** component set up to visualize muscle-related

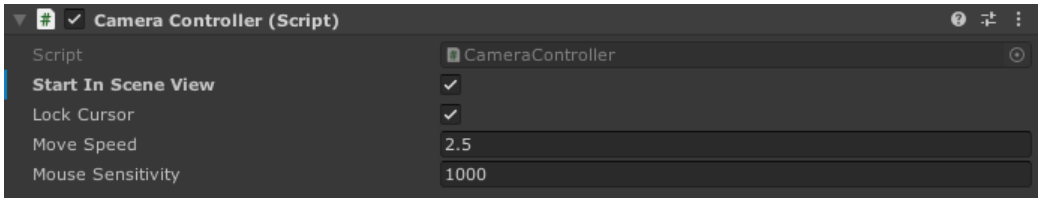
data in Scene view, but you can also use the provided free-roam camera (“Main Camera”) to navigate around the Game view when Play Mode is active.

To switch between whether Play Mode starts in Game view or Scene view, select “Main Camera” in the Hierarchy window and toggle *Start in Scene View* under the **Camera Controller** component.

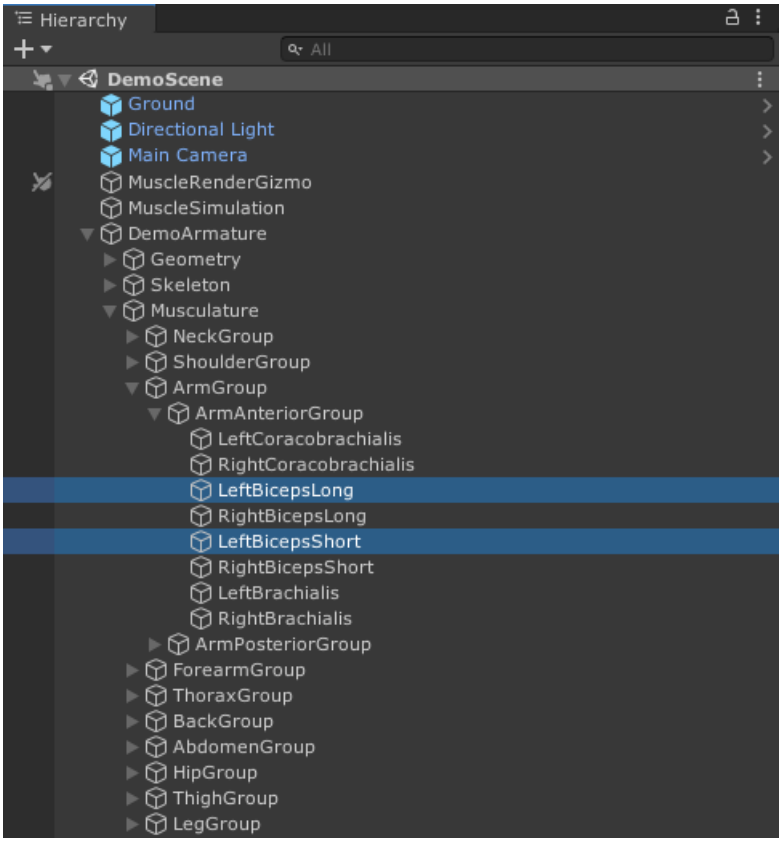
Interacting with the Demo

The demo has a **Muscle Simulation** component attached to each muscle that allows you to adjust muscle excitation manually. Here’s a quick step through:

1. Select “Main Camera” in the Hierarchy window and navigate to the **Camera Controller** component in the Inspector window. Make sure *Start in Scene View* is checked.



2. In the Hierarchy window, expand “DemoArmature” > “Musculature” > “ArmGroup” > “ArmAnteriorGroup.” You should see the list of arm flexor muscles including the biceps. Hold Ctrl and select both the “LeftBicepsLong” and “LeftBicepsShort” muscles. You should see both muscles highlighted in the Scene view.



3. Start Play Mode by selecting the Play icon or Ctrl + P.
4. In the Inspector window, navigate to the **Muscle Stimulator** component and adjust the *Excitation* slider. Watch the muscles react in Scene view.
5. Play around with other muscles. You can also see what happens if you adjust parameters such as *Normalized SEE Slack Length* and *Max Isometric Force*, though you'll want to stop Play Mode before changing parameter values.

Tips & Tricks

Tips

- You can change the excitation of multiple muscles together by selecting them simultaneously and adjusting the the *Excitation* field slider under the **Muscle Stimulator** component.
- When attempting to work on specific muscles in isolation, it sometimes helps to pin **Rigidbody** components you're not working with by setting their *Is Kinematic* fields to `true`.

Caveats

- Muscles still exert passive effects even without activation, so joint torques still need be calculated for all muscles regardless of activation level to get the full effect.
- The `muscleNodes` field in `MuscleTendonUnit` expects muscle nodes to be defined *in order from muscle origin to insertion*. This ordering implicitly encodes the direction of muscle contraction (which begins with the insertion and moves toward the origin).
- For those from a more traditional academic background who might be unfamiliar, Unity uses a left-handed coordinate system. Keep in mind that rotations therefore follow the same principle.
- Try not to assign too much meaning to any particular paramater value or put too much stock in finding real-world reference values. The parameters and configurations you use are what they are and will be what they need to be.

Troubleshooting Muscle Movement

This is a big topics since Unity physics has a lot of parts to it.**Rigidbody**, **Joint**, **Muscle Node**, and **Muscle Tendon Unit** configurations all affect how bodies and muscles behave, and there are myriad reasons why muscles may fail to behave as anticipated (beyond your own misconceptions). I've created an initial checklist from issues that I've encountered:

- Make sure that each muscle node `Bone` field references the actual `GameObject` that you want to attach to. Errors can easily be made if you use the Object Picker interface and have objects that share the same name.
- Following from the above, do a check in the Scene View from all angles that your muscle nodes are in their intended positions. The muscle torques being generated can be very sensitive to muscle node placement.
- A muscle node configuration that behaves acceptably in one body position can produce unintended movement once rotated far enough into another configuration. You can of course run simulations and observe the resulting behaviors, but it can help to first inspect muscles while rotating limbs around in the Scene View to check that they won't enter an obviously undesirable configuration.
- Check the Unity Joint and ragdoll stability page.

- Ensure that the global 3D physics settings are set as you want them, such as *Default Max Angular Speed*. To find these settings, navigate to *Edit > Project Settings > Physics*.
- If your rigid bodies seem a bit too sensitive or a bit too resistive to movement under muscle activation, you may need to manually adjust their inertia tensors.
- Check the configurations on the **Rigidbody** components. Make sure that:
    - The *Mass* value isn't too high relative to the maximum isometric force of the muscles moving the rigid body.
    - *Drag* and *Angular Drag* aren't set too high. Otherwise, the muscles will have trouble overcoming the movement resistance.
    - *Is Kinematic* is not checked. If it is, physics won't affect the rigid body.
    - *Use Gravity* is set as intended. Muscles behave very differently under gravity versus without. Turning off gravity on rigid bodies can be useful sometimes for debugging, but don't forget the context of your results.
    - Joint constraints are set as intended. Make sure you didn't accidentally leave any of the *Freeze Position* or *Freeze Rotation* coordinate-axes flags on if you didn't intend to.
- Check the configurations of the **Joint** components involved in the motion. In particular:
    - Make sure that the angular limits allow or restrict the movements that you expect to be allowed or restricted.
    - Double-check that the *Connected Body* field is properly set on each **Joint** component.
- If you're using *Is Kinematic* to pin rigid bodies while testing, be aware that motions may involve a broader combination of muscles, bodies, and joints than you might understand. Restricting movement in specific parts can effect movement in adjacent areas.
- On muscles, check the values for the following fields:
    - *Normalized SEE Slack Length* affects the length ratio of the "passive" serial elastic element to the "active" contractile element. Setting this value higher makes the muscle more elastic and less responsive, potentially getting more noticeable oscillatory behavior. This also makes the muscle's passive effects more pronounced, so a muscle with a high normalized SEE slack length value can powerfully inhibit the movement of antagonist muscles.
    - *Max Isometric Force* affects the overall magnitude of the torques generated by the muscle. If this value is set too low, the muscle will be ineffective in driving motion. If this value is set too high, it's usually pretty obvious.

Questions & Discussion

Questions

Why create Kinesis?

Kinesis was created mostly as a byproduct of attempting to create a base for further AI experiments in embodied cognition. I wanted an agent with a complete physical presence that could navigate and interact with a broader physical environment in an organic and "non-scripted" way. Although I didn't think it would be a cakewalk, I simply misjudged the amount of work it would take to both implement and work with such a muscle-based system (at least, for someone like me without a graduate background in biodynamics, robotics, or mathematics). In hindsight, I could've picked a better approach for what I wanted to accomplish, but working on Kinesis was rewarding in its own right.

The reason I decided to clean up the implementation and release Kinesis on the Unity Asset Store was because I didn't really find anything like it available for Unity, and it definitely would've been nice to have something like it when starting out. My hope is that fellow experimenters find it a useful starting point.

What does the muscle model in Kinesis ignore and what is it missing?

The muscle model in Kinesis is a somewhat simpler version of Hill-type muscle model. While it does handle segmented pathing, features that were not included in the initial implementation include:

- Modeling of muscle fiber pennation angles.
- Modeling of serial damping effects.
- Handling for more complex muscles such as those with multiple heads, insertions, and sheet-like muscles.

There are other phenomena that Hill-type muscles models don't typically seem to address:

- Muscle mass, although there has been at least some exploration of this with Hill-type muscle models.
- The effects of previous activity (of various timescales) on muscle dynamics.

The model applies forces using the muscle segments as lines of action. Why use `AddTorque()` ***instead of*** `AddForce()`**?**

I did experiment with `AddForce()` in the beginning, but it quickly became apparent that the approach was inherently unstable, even though one would imagine they should thereotically generate the same results. Not only was using `AddTorque()` more stable, but it also simplified the code required for movement forces.

Contact

General Questions and Comments: contact@squeakyspacebar.com

Support Questions: support@squeakyspacebar.com

Versioning

Kinesis attempts to follow semantic versioning rules.

Changelog

- v1.0.0
  - Initial release.

Bibliography

1. Geijtenbeek, T., van der Stappen, A. F., & Panne, M. (2013). Flexible Muscle-Based Locomotion for Bipedal Creatures. *ACM Transactions on Graphics, 32*(6), 206:1–206:11. https://doi.org/10.1145/2508363.2508399.
2. Hill, A.W. (1938). The heat of shortening and the dynamic constants of muscle. *Proceedings of the Royal Society of London Series B, 126*(843), 136–195.
3. Zhou, J., Chen, J., Deng, H., & Qiao, H. (2019). From rough to precise: human-inspired phased target learning framework for redundant musculoskeletal systems. *Frontiers in Neurorobotics*, *13*. https://doi.org/10.3389/fnbot.2019.00061

Further Reading

If you're looking to understand the implementation of Hill-type muscle models in a general way (Zajac, 1989) is a good paper to understand. (Günther, et al., 2012) is an interesting treatment of adding mass to Hill models. The rest of these are resources for approaches to muscle-based motor control.

1. Dongsung H. & Todorov, E. (2009). *Real-time motor control using recurrent neural networks*. 2009 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL). Nashville, TN, USA. https://doi.org/10.1109/ADPRL.2009.4927524
2. Driess, D., Zimmermann, H., Wolfen, S., Suissa, D., Haeufle, D., Hennes, D., Toussaint, M., & Schmitt, S. (2018). *Learning to control redundant musculoskeletal systems with neural networks and SQP: exploiting muscle properties*. 2018 IEEE International Conference on Robotics and Automation (ICRA). Brisbane, QLD, Australia. https://doi.org/10.1109/icra.2018.8463160
3. Günther M., Röhrle O., Haeufle D. F., & Schmitt S. (2012). Spreading out muscle mass within a Hill-type model: a computer simulation study. *Computational and Mathematical Methods in Medicine*, 2012. https://doi.org/10.1155/2012/848630
4. Rückert, E.A., & d'Avella, A. (2013). Learned parametrized dynamic movement primitives with shared synergies for controlling robotic and musculoskeletal systems. *Frontiers in Computational Neuroscience, 7*. https://doi.org/10.3389/fncom.2013.00138
5. Wochner, I., Driess, D., Zimmermann, H., Haeufle, D. F. B., Toussaint, M., & Schmitt, S. (2020). Optimality principles in human point-to-manifold reaching accounting for muscle dynamics. *Frontiers in Computational Neuroscience*, 14. https://doi.org/10.3389/fncom.2020.00038
6. Zajac F. E. (1989). Muscle and tendon: properties, models, scaling, and application to biomechanics and motor control. *Critical reviews in biomedical engineering*, *17*(4), 359–411.