Unity Plug-in

Introduction

The MuJoCo Unity plug-in allows the Unity Editor and runtime to use the MuJoCo physics engine. Users can import MJCF files and edit the models in the Editor. The plug-in relies on Unity for most aspects – assets, game logic, simulation time – but uses MuJoCo to determine how objects move, giving the designer access to MuJoCo's full API.

Installation instructions

The plug-in directory (available at https://github.com/deepmind/mujoco/tree/main/unity) includes a `package.json` file. Unity's package manager recognizes this file and will import the plug-in's C# codebase to your project. In addition, Unity also needs the native MuJoCo library, which can be found in the specific platform archive at https://github.com/deepmind/mujoco/releases.

On Unity version 2020.2 and later, the Package Manager will look for the native library file and copy it to the package directory when the package is imported. Alternatively, you can manually copy the native library to the package directory and rename it, see platform-specific instructions below. The library can also be copied into any location under your project's Assets directory.

MacOS

The MuJoCo app needs to be run at least once before the native library can be used, in order to register the library as a trusted binary. Then, copy the dynamic library file from `/Applications/MuJoCo.app/Contents/Frameworks/mujoco.framework/Versions/Current/libmujoco.2.3.1.dylib` (it can be found by browsing the contents of `MuJoCo.app`) and rename it as `mujoco.dylib`.

Linux

Expand the `tar.gz` archive to `~/.mujoco`. Then copy the dynamic library from `~/.mujoco/mujoco-2.3.1/lib/libmujoco.so.2.3.1` and rename it as `libmujoco.so`.

Windows

Expand the `zip` archive to a directory called `MuJoCo` in your user directory, and copy the file `MuJoCo\bin\mujoco.dll`.

Using the plug-in

Importer

The importer is invoked from the Editor's *Asset* menu: click on "Import MuJoCo Scene" and select the XML file with your model's MJCF specification.

Context menus

- Right-clicking a geom component offers two options:
  - "Add mesh renderer" adds components to the same game object that render the geom: a standard `MeshRenderer` and a `MjMeshFilter` that creates a procedural mesh that is recreated when the geom shape properties change.
  - "Convert to free object" adds two new game objects: a parent with an `MjBody` component and a sibling with an `MjFreeJoint` component. This allows the previously static geom to move about freely in the scene. This action only applies to "world" geoms – those that do not currently have an `MjBody` parent.
- Right-clicking a Unity Collider offers the option to "Add a matching MuJoCo geom" to the same game object. Note that this does not comprise a complete conversion of the physics – Rigidbody, ArticulationBody and Joint configurations still need to be recreated manually.

Mouse spring

When the selected game object has an `MjBody` component, spring forces can be applied to this body towards the mouse cursor through a control-left-drag action in the Scene view. The 3D position of the spring force origin is found by projecting the mouse position on a plane defined by the camera X direction and the world Y direction. Adding the shift key changes the projection plane to be parallel to the world's X and Z axes.

Tips to Unity users

- If any compilation or runtime errors are encountered, the state of the system is undefined. Therefore, we recommend turning on "Error Pause" in the console window.
- In PhysX, every *Rigidbody* is a "free body". In contrast, MuJoCo requires explicit specification of joints for mobility. For convenience, we provide a context menu for "freeing" a world geom (i.e., an `MjGeom` component without any `MjBody` ancestor) by adding a parent `MjBody` and a sibling `MjFreeJoint`.
- The plug-in doesn't support collision detection without physical presence, so there is no built-in notion of trigger colliders. The presence or absence of a contact force can be read by adding a touch sensor and reading its `SensorReading` value (which will correspond to the normal force, see *touch sensor documentation <sensor-touch>*).

Design principles

The plug-in design provides a one-to-one mapping between MJCF elements and Unity components. In order to simulate a Unity scene (e.g., when the user hits the "play" button in the

Editor) using MuJoCo, the plug-in:

1. Scans the GameObject hierarchy in the scene for MuJoCo components.
2. Creates an MJCF description and passes it to MuJoCo's compiler.
3. Binds every component to the MuJoCo runtime via the corresponding index in MuJoCo's data structures. This index is used for updating Unity's transforms during simulation.

This design principle has several implications:

- Most fields of the Unity components correspond directly to MJCF attributes. Therefore, the user can refer to the MuJoCo documentation for details on the semantics of different values.
- The layout of MuJoCo components in the GameObject hierarchy determines the layout of the resulting MuJoCo model. Therefore, we adopt a design rule that **every game object must have at most one MuJoCo component**.
- We rely on Unity for spatial configuration, which requires vector components to be swizzled since Unity uses left-handed frames with Y as the vertical axis, while MuJoCo uses right-handed frames with Z as the vertical axis.
- Unity transform scaling affects positions, orientations, and scale of the entire game object subtree. However, MuJoCo doesn't support collision of skewed cylinders and capsules (skewed spheres are supported via the ellipsoid primitive). The gizmo for geoms and sites ignores this skew (similarly to PhysX colliders), and will always show the primitive shape as it will appear to the physics.
- During runtime, changing values of component fields will not trigger scene recreation, so it will have no immediate effect on the physics. However, the new values will be loaded upon the next scene recreation.

Wherever possible, we do things the Unity Way: gravity is read from Unity's physics settings, and the simulation step is read from Unity's Time Manager's *Fixed Timestep*. All aspects of appearance (e.g., meshes, materials, and textures) are handled by Unity's Asset Manager, and RGBA specifications are done using material assets.

Implementation notes

Importer workflow

When the user selects an MJCF file, the importer first loads the file in MuJoCo, saves it to a temporary location, and then processes the generated saved file. This has several effects:

- It validates the MJCF - we are guaranteed that the saved MJCF matches the schema.
- It validates the assets (materials, meshes, textures) and imports these assets into Unity, as well as creating new material assets for geom RGBA specification.
- It allows the importer to handle <include> elements without replicating MuJoCo's file-system workflow.
- The current version of MuJoCo generates MJCF files with explicit <inertial> elements, even when the original model uses geoms for implicit definition of the body inertia. If you plan to change geom properties of an imported model, remove these auto-generated `MjInertial` components manually. We plan to address this in a future release of MuJoCo.

In Unity, there is no equivalent to MJCF's "cascading" <default> clauses. Therefore, components in Unity reflect the corresponding elements' state after applying all the relevant default classes, and the class structure in the original MJCF is discarded.

The MuJoCo Scene

When a MuJoCo scene is created, the `MjScene` component first scans the scene for all instances of `MjComponent`. Each component creates its own MJCF element using Unity scene's spatial structure to describe the model's initial reference pose (called `qpos0` in MuJoCo). `MjScene` combines these XML elements according to the hierarchy of the respective game objects and creates a single MJCF description of the physics model. It then creates the runtime structs `mjModel` and `mjData`, and binds each component to the runtime by identifying its unique index.

During runtime, `MjScene.FixedUpdate()` calls mj_step, and then synchronizes the state of each game object according to the index `MjComponent.MujocoId` identified at binding time. An `MjScene` component is added automatically when the application starts (e.g., when the user hits "play") if and only if the scene includes any MuJoCo components. If your application's initialization phase involves ticking the physics while adding game objects and components, you can call `MjScene.CreateScene()` when the initialization phase is over.

Scene recreation maintains continuity of physics and state in the following way:

1. The position and velocity of joints are cached.
2. MuJoCo's state is reset (to `qpos0`) and Unity transforms are synchronized.
3. A new XML is generated, creating a model that has the same `qpos0` as the previous one for the joints that persisted.
4. The MuJoCo state (for the joints that persisted) is set from the cache, and Unity transforms are synchronized.

Because the MuJoCo library doesn't (yet) expose an API for scene editing, adding and removing MuJoCo components causes complete scene recreation. This can be expensive for large models or if it happens frequently. We expect this performance limitation to be lifted in future versions of MuJoCo.

Global Settings

An exception to the one-element-per-one-component is the Global Settings component. This component is responsible for all the configuration options that are included in the fixed-size, singleton, global elements of MJCF. Currently it holds information that corresponds to the <option> and <size> elements, and in the future it will also be used for the <compiler> element, if/when fields there will be relevant to the Unity plug-in.

Invoking the importer at application runtime

The importer is implemented by the class `MjImporterWithAssets`, which is a subclass of `MjcfImporter`. This parent class takes an MJCF string and generates the hierarchy of components. It

can be invoked at play-time (it doesn't involve Editor functionality), and it doesn't invoke any functions of the MuJoCo library. This is useful when MuJoCo models are generated procedurally (e.g., by some evolutionary process) and/or when an MJCF is imported only to be converted (e.g., to PhysX, or URDF). Since it cannot interact with Unity's `AssetManager` (which is a feature of the Editor), this class's functionality is restricted. Specifically:

- It ignores all assets (including collision meshes).
- It ignores visuals (including RGBA specifications).

MuJoCo sensor components

MuJoCo defines many sensors, and we were concerned that creating a separate `MjComponent` class for each would lead to a lot of code duplication. Therefore, we created classes according to the type of object (actuator / body / geom / joint / site) whose properties are measured, and the type (scalar / vector / quaternion) of the measured data.

Here's a table that maps types to sensors:

Mujoco Object Type	Data Type	Sensor Name
Actuator	Scalar	• `actuatorpos` • `actuatorvel` • `actuatorfrc`
Body	Vector	• `subtreecom` • `subtreelinvel` • `subtreeangmom` • `framepos` • `framexaxis` • `frameyaxis` • `framezaxis` • `framelinvel` • `frameangvel` • `framelinacc` • `frameangacc`
Body	Quaternion	• `framequat`
Geom	Vector	• `framepos` • `framexaxis` • `frameyaxis` • `framezaxis` • `framelinvel` • `frameangvel` • `framelinacc` • `frameangacc`
Geom	Quaternion	• `framequat`
Joint	Scalar	• `jointpos` • `jointvel` • `jointlimitpos` • `jointlimitvel` • `jointlimitfrc`
Site	Scalar	• `touch` • `rangefinder`
Site	Vector	• `accelerometer` • `velocimeter` • `force` • `torque` • `gyro` • `magnetometer` • `framepos` • `framexaxis` • `frameyaxis` • `framezaxis` • `framelinvel` • `frameangvel` • `framelinacc` • `frameangacc`
Site	Quaternion	• `framequat`

Here's the same table in reverse, mapping sensors to classes:

Sensor Name	Plugin Class
`accelerometer`	SiteVector
`actuatorfrc`	ActuatorScalar
`actuatorpos`	ActuatorScalar
`actuatorvel`	ActuatorScalar
`force`	SiteVector
`frameangacc`	*Vector (depends on frame type)
`frameangvel`	*Vector (depends on frame type)

Sensor Name	Plugin Class
`framelinacc`	*Vector (depends on frame type)
`framelinvel`	*Vector (depends on frame type)
`framepos`	*Vector (depends on frame type)
`framequat`	*Quaternion (depends on frame type)
`framexaxis`	*Vector (depends on frame type)
`frameyaxis`	*Vector (depends on frame type)
`framezaxis`	*Vector (depends on frame type)
`gyro`	SiteVector
`jointlimitfrc`	JointScalar
`jointlimitpos`	JointScalar
`jointlimitvel`	JointScalar
`jointpos`	JointScalar
`jointvel`	JointScalar
`magnetometer`	SiteVector
`subtreeangmom`	BodyVector
`subtreecom`	BodyVector
`subtreelinvel`	BodyVector
`torque`	SiteVector
`touch`	SiteScalar
`velocimeter`	SiteVector

The following sensors are not yet implemented:

```
tendonpos
tendonvel
ballquat
ballangvel
tendonlimitpos
tendonlimitvel
tendonlimitfrc
user
```

Mesh Shapes

The plug-in allows using arbitrary Unity meshes for MuJoCo collision. At model compilation, MuJoCo calls qhull to create a convex hull of the mesh, and uses that for collisions. Currently the computed convex hull is not visible in Unity, but we intend to expose it in future versions.

Interaction with External Processes

Roboti's MuJoCo plug-in for Unity steps the simulation in an external Python process, and uses Unity only for rendering. In contrast, our plug-in relies on Unity to step the simulation. It should be possible to use our plug-in while an external process "drives" the simulation, for example by seting `qpos`, calling `mj_kinematics`, synchronizing the transforms, and then using Unity to render or compute game logic. In order to establish communication with an external process, you can use Unity's ML-Agents package.