Model Editing

> New API
>
> The API described below is new but feature complete. It is recommended for general use, but latent bugs are still possible. Please report any issues on GitHub.

As of MuJoCo 3.2.0, it is possible to create and modify models using the mjSpec struct and related API. This datastructure is in one-to-one correspondence with MJCF and indeed, MuJoCo's own XML parsers (both MJCF and URDF) use this API when loading a model.

Overview

The new API augments the traditional workflow of creating and editing models using XML files, breaking up the *parse* and *compile* steps. As summarized in the Overview chapter, the traditional workflow is:

1. Create an XML model description file (MJCF or URDF) and associated assets.
2. Call mj_loadXML, obtain an mjModel instance.

The new workflow using mjSpec is:

1. Create an empty mjSpec or parse an existing XML file.
2. Programmatically edit the mjSpec datastructure by adding, modifying and removing elements.
3. Compile the mjSpec to an mjModel instance.

After compilation, the mjSpec remains editable, so steps 2 and 3 are interchangeable.

Usage

Here we describe the C API for procedural model editing, but it is also exposed in the Python bindings. Advanced users can refer to user_api_test.cc and the MJCF parser in xml_native_reader.cc for more usage examples. After creating a new mjSpec or parsing an existing XML file to an mjSpec, procedural editing corresponds to setting attributes. For example, in order to change the timestep, one can do:

```
mjSpec* spec = mj_makeSpec();
spec->opt.timestep = 0.01;
...
mjModel* model = mj_compile(spec, NULL);
```

Attributes which have variable length are C++ vectors and strings, exposed to C as opaque types. In C one uses the provided getters and setters:

```
mjs_setString(model->modelname, "my_model");
```

In C++, one can use vectors and strings directly:

```
std::string modelname = "my_model";
*spec->modelname = modelname;
```

Loading a spec from XML can be done as follows:

```
std::array<char, 1000> error;
mjSpec* s = mj_parseXML(filename, vfs, error.data(), error.size());
```

Model elements

Model elements corresponding to MJCF are exposed to the user as C structs with the `mjs` prefix, the definitions are listed under the Model Editing section of the struct reference. For example, an MJCF geom corresponds to an mjsGeom.

Global defaults for all elements are set by initializers like mjs_defaultGeom. These functions are defined in user_init.c and are the source of truth for all default values.

Elements cannot be created directly; they are returned to the user by the corresponding constructor function, e.g. mjs_addGeom. For example, to add a box geom to the world body, one would do

```
mjSpec* spec = mj_makeSpec();                                  // make an empty spec
mjsBody* world = mjs_findBody(spec, "world");                  // find the world body
mjsGeom* my_geom = mjs_addGeom(world, NULL);                   // add a geom to the world
my_geom->type = mjGEOM_BOX;                                    // set geom type
my_geom->size[0] = my_geom->size[1] = my_geom->size[2] = 0.5;  // set box
mjModel* model = mj_compile(spec, NULL);                       // compile
```

The `NULL` second argument to mjs_addGeom is the optional default class pointer. When using defaults procedurally, default classes are passed in explicitly to element constructors. The global defaults of all elements (used when no default class is passed in) can be inspected in user_init.c.

Attachment

This framework introduces a powerful new feature: attaching and detaching model subtrees. This feature is already used to power the attach an replicate meta-elements in MJCF. Attachment allows the user to move or copy a subtree from one model into another, while also copying or moving related referenced assets and referencing elements from outside the kinematic tree (e.g., actuators and sensors). Similarly, detaching a subtree will remove all associated elements from the model. The default behavior is to move the child into the parent while attaching, so subsequent changes to the child will also change the parent. Alternatively, the user can choose to make an entirely new copy during attach using mjs_setDeepCopy. This flag is temporarily set to true while parsing XMLs. It is possible to attach a body to a frame:

```
mjSpec* parent = mj_makeSpec();
mjSpec* child = mj_makeSpec();
parent->compiler.degree = 0;
child->compiler.degree = 1;
mjsElement* frame = mjs_addFrame(mjs_findBody(parent, "world"), NULL)->element;
mjsElement* body = mjs_addBody(mjs_findBody(child, "world"), NULL)->element;
mjsBody* attached_body_1 = mjs_asBody(mjs_attach(frame, body, "attached-", "-1"));
```

or attach a body to a site:

```
mjSpec* parent = mj_makeSpec();
mjSpec* child = mj_makeSpec();
mjsElement* site = mjs_addSite(mjs_findBody(parent, "world"), NULL)->element;
mjsElement* body = mjs_addBody(mjs_findBody(child, "world"), NULL)->element;
mjsBody* attached_body_2 = mjs_asBody(mjs_attach(site, body, "attached-", "-2"));
```

or attach a frame to a body:

```
mjSpec* parent = mj_makeSpec();
mjSpec* child = mj_makeSpec();
mjsElement* body = mjs_addBody(mjs_findBody(parent, "world"), NULL)->element;
mjsElement* frame = mjs_addFrame(mjs_findBody(child, "world"), NULL)->element;
mjsFrame* attached_frame = mjs_asFrame(mjs_attach(body, frame, "attached-", "-1"));
```

Note that in the above examples, the parent and child models have different values for `compiler.degree`, corresponding to the compiler/angle attribute, speci which angles are interperted. Compiler flags are carried over during attachment, so the

child model will be compiled using the child flags, while the parent will be compiled using the parent flags.

Note also that once a child is attached by reference to a parent, the child cannot be compiled on its own.

Default classes

Default classes are fully supported in the new API, however using them requires an understanding of how defaults are implemented. As explained in the Default settings section, default classes are first loaded as a tree of dummy elements, which are then used to initialize elements which reference them. When editing models with defaults, this initialization is explicit:

```
mjSpec* spec = mj_makeSpec();
mjsDefault* main = mjs_getSpecDefault(spec);
main->geom.type = mjGEOM_BOX;
mjsGeom* geom = mjs_addGeom(mjs_findBody(spec, "world"), main);
```

Importantly, changing a default class after it has been used to initialize elements will not change the properties of already initialized elements.

> **Possible future change**
>
> The behaviour described above, where defaults are only applied at initialization, is a remnant of the old, XML-only loading pipeline. A future API change could allow defaults to be changed and applied after initialization. If you think this feature is important to you, please let us know on GitHub.

XML saving

Specs can be saved to an XML file or string using mj_saveXML or mj_saveXMLString, respectively. Saving requires that the spec first be compiled. Importantly, the saved XML will take into account any defined defaults. This is useful when a model has many repeated values, for example if loaded from URDF, which does not support defaults. In such a case one can add default classes, set the class of the relevant elements, and save; the resulting XML will use the defaults and be more human-readable.

In-place recompilation

Compilation with mj_compile can be called at any point to obtain a new mjModel instance. In contrast, mj_recompile updates an existing mjModel and r stable place, while preserving the simulation state. This allows model editing to occur **during simulation**, for example adding or removing bodies.