Code samples

MuJoCo comes with several code samples providing useful functionality. Some of them are quite elaborate (simulate.cc in particular) but nevertheless we hope that they will help users learn how to program with the library.

testspeed

This code sample times the simulation of a given model. The timing is straightforward: the simulation of the passive dynamics (with optional control noise) is rolled-out for the specified number of steps, while collecting statistics about the number of contacts, scalar constraints, and CPU times from internal profiling. The results are then printed to the console. To simulate controlled dynamics instead of passive dynamics one can either install a control callback mjcb_control, or modify the code to set control signals explicitly, as explained in the simulation loop section below. This command-line utility is run with

```
testspeed modelfile [nstep nthread ctrlnoise npoolthread]
```

Where the command line arguments are

Argument	Default	Meaning
`modelfile`	(required)	path to model
`nstep`	10000	number of steps per rollout
`nthread`	1	number of threads running parallel rollouts
`ctrlnoise`	0.01	scale of pseudo-random noise injected into actuators
`npoolthread`	1	number of threads in engine-internal threadpool

Notes:

- When `nthread > 1` is specified, the code allocates a single mjMo
thread mjData, and runs `nthread` identical simulations in parallel.
performance with all cores active, as in Reinforcement Learning scenarios where

stable

samples are collected in parallel. The optimal `nthread` usually equals the number of logical cores.

- By default, the simulation starts from the model reference configuration with zero velocities. However, if a keyframe named "test" is present in the model, it is used as the initial state.
- The `ctrlnoise` argument prevents models from settling into a static state where, due to warmstarts, one can measure artificially faster simulation.
- When `npoolthread > 1` is specified, an engine-internal mjThreadPool is created with the specified number of threads, to speed up simulation of large scenes. Note that while it is possible to use both `nthread` and `npoolthread`, the scenarios for which one would want these different types of multithreading are usually mutually exclusive.
- For more repeatable performance statistics, run the tool with the `performance` governor on Linux, or the `High Performance` power plan on Windows, to reduce noise from CPU scaling.
- Many modern CPUs contain a mixture of "performance" and "efficiency" cores. Users should consider restricting the process to only run on the same type of cores for more interpretable performance statistics. This can be done via the taskset command on Linux, or the start /affinity command on Windows (processor affinity cannot be specified through documented API means on macOS).

simulate

This code sample is a fully-featured interactive simulator. It opens an OpenGL window using the platform-independent GLFW library, and renders the simulation state in it. There is built-in help, simulation statistics, profiler, sensor data plots. The model file can be specified as a command-line argument, or loaded at runtime using drag-and-drop functionality. This code sample uses the native UI to render various controls, and provides an illustration of how the new UI framework is intended to be used. Below is a screen-capture of `simulate` in action:

stable

Interaction is done with the mouse; built-in help with a summary of available commands is available by pressing the `F1` key. Briefly, an object is selected by left-double-click. The user can then apply forces and torques on the selected object by holding Ctrl and dragging the mouse. Dragging the mouse alone (without Ctrl) moves the camera. There are keyboard shortcuts for pausing the simulation, resetting, and re-loading the model file. The latter functionality is very useful while editing the model in an XML editor.

The code is long yet reasonably commented, so it is best to just read it. Here we provide a high-level overview. The `main()` function initializes both MuJoCo and GLFW, opens a window, and install GLFW callbacks for mouse and keyboard handling. Note that there is no render callback; GLFW puts the user in charge, instead of running a rendering loop behind the scenes. The main loop handles UI events and rendering. The simulation is handled in a background thread, which is synchronized with the main thread.

The mouse and keyboard callbacks perform whatever action is necessary. Many of these actions invoke functionality provided by MuJoCo's abstract visualization mechanism. Indeed this mechanism is designed to be hooked to mouse and keyboard events more or less directly, and provides camera as well as perturbation control.

The profiler and sensor data plots illustrate the use of the mjr_figure function that can plot elaborate 2D figures with grids, annotation, axis scaling etc. The information presented in the profiler is extracted from the diagnostic fields of mjDa stable useful tool for tuning the parameters of the constraint solver algorithms. The outputs of the sensors defined in the model are visualized as a bar graph.

Note that the profiler shows timing information collected with high-resolution timers. On Windows, depending on the power settings, the OS may reduce the CPU frequency; this is because simulate.cc sleeps most of the time in order to slow down to realtime. This results in inaccurate timings. To avoid this problem, change the Windows power plan so that the minimum processor state is 100%.

compile

This code sample evokes the built-in parser and compiler. It implements all possible model conversions from (MJCF, URDF, MJB) format to (MJCF, MJB, TXT) format. Models saved as MJCF use a canonical subset of our format as described in the Modeling chapter, and therefore MJCF-to-MJCF conversion will generally result in a different file. The TXT format is a human-readable road-map to the model. It cannot be loaded by MuJoCo, but can be a very useful aid during model development. It is in one-to-one correspondence with the compiled mjModel. Note also that one can use the function mj_printData to create a text file which is in one-to-one correspondence with mjData, although this is not done by the code sample.

basic

This code sample is a minimal interactive simulator. The model file must be provided as command-line argument. It opens an OpenGL window using the platform-independent GLFW library, and renders the simulation state at 60 fps while advancing the simulation in real-time. Press Backspace to reset the simulation. The mouse can be used to control the camera: left drag to rotate, right drag to translate in the vertical plane, shift right drag to translate in the horizontal plane, scroll or middle drag to zoom.

The Visualization programming guide below explains how visualization works. This code sample is a minimal illustration of the concepts in that guide.

record

This code sample simulates the passive dynamics of a given model, renders it offscreen, reads the color and depth pixel values, and saves them into a raw data file that can then be converted into a movie file with tools such as ffmpeg. stable
simplified compared to simulate.cc because there is no user interaction, visualization options or timing; instead we simply render with the default settings as fast as

possible. The dimensions and number of multi-samples for the offscreen buffer are specified in the MuJoCo model with the visual/global/{offwidth, offheight} and visual/quality/offsamples attributes, while the simulation duration, frames-per-second to be rendered (usually much less than the physics simulation rate), and output file name are specified as command-line arguments.

```
record modelfile duration fps rgbfile [adddepth]
```

Where the command line arguments are

Argument	Default	Meaning
`modelfile`	(required)	path to model
`duration`	(required)	duration of the recording in seconds
`fps`	(required)	number of frames per second
`rgbfile`	(required)	path to raw recording file
`adddepth`	1	overlay depth image in the lower left corner (0: none)

For example, a 5 second animation at 60 frames per second is created with:

```
record humanoid.xml 5 60 rgb.out
```

The default humanoid.xml model specifies offscreen rendering with 2560x1440 resolution. With this information in hand, we can compress the (large) raw data file into a playable movie file:

```
ffmpeg -f rawvideo -pixel_format rgb24 -video_size 2560x1440
       -framerate 60 -i rgb.out -vf "vflip,format=yuv420p" video.mp4
```

Note that the offscreen rendering resolution of the model and ffmpeg's video_size must be identical.

This sample can be compiled in three ways which differ in how the OpenGL context is created: using GLFW with an invisible window, using OSMesa, or using EGL. The latter two options are only available on Linux and are envoked by defining the symbols MJ_OSMESA or MJ_EGL when compiling record.cc. The functions `initOpenGL` and `closeOpenGL` create and close the OpenGL context in three different ways depending on which of the above symbols is defined.

Note that the MuJoCo rendering code does not depend on how the Op stable was created. This is the beauty of OpenGL: it leaves context creation t and the actual rendering is then standard and works in the same way on all platforms.

In retrospect, the decision to leave context creation out of the standard has led to unnecessary proliferation of overlapping technologies, which differ not only between platforms but also within a platform in the case of Linux. The addition of a couple of extra functions (such as those provided by OSMesa for example) could have avoided a lot of confusion. EGL is a newer standard from Khronos which aims to do this, and it is gaining popularity. But we cannot yet assume that all users have it installed.

stable