Visualization

MuJoCo has a native 3D visualizer. Its use is illustrated in the simulate.cc code sample and in the simpler basic.cc code sample. While it is not a full-featured rendering engine, it is a convenient, efficient and reasonably good-looking visualizer that facilitates research and development. It renders not only the simulation state but also decorative elements such as contact points and forces, equivalent inertia boxes, convex hulls, kinematic trees, constraint violations, spatial frames and text labels; these can provide insight into the physics simulation and help fine-tune the model.

The visualizer is tightly integrated with the simulator and supports both onscreen and offscreen rendering, as illustrated in the record.cc code sample. This makes it suitable for synthetic computer vision and machine learning applications, especially in cloud environments. VR integration is also available as of MuJoCo version 1.40, facilitating applications that utilize new head-mounted displays such as Oculus Rift and HTC Vive.

Visualization in MuJoCo is a two-stage process:

Abstract visualization and interaction
: This stage populates the mjvScene data structure with a list of geometric objects, lights, cameras and everything else needed to produce a 3D rendering. It also provides abstract keyboard and mouse hooks for user interaction. The relevant data structure and function names have the prefix `mjv`.

OpenGL rendering
: This stage takes the mjvScene data structure populated in the abstract visualization stage, and renders it. It also provides basic 2d drawing and framebuffer access, so that most applications would not need to call OpenGL directly. The relevant data structure and function names have the prefix `mjr`.

There are several reasons for this separation. First, the two stages are conceptually different and separating them is good software design. Second, they have different dependencies, both internally and in terms of additional libraries; in particular, abstract visualization does not require any graphics libraries. Third, users who wish to integrate another rendering engine with MuJoCo can bypass the native OpenGL renderer but still take advantage of the abstract visualizer.

Below is a mixture of C code and pseudo-code in comments, illustratir stable
of a MuJoCo application which does both simulation and rendering. This is a short version of the basic.cc code sample. For concreteness we assume that GLFW is used,

although it can be replaced with a different window library such as GLUT or one of its derivatives.

```
// MuJoCo data structures
mjModel* m = NULL;                  // MuJoCo model
mjData* d = NULL;                   // MuJoCo data
mjvCamera cam;                      // abstract camera
mjvOption opt;                      // visualization options
mjvScene scn;                       // abstract scene
mjrContext con;                     // custom GPU context

// ... load model and data

// init GLFW, create window, make OpenGL context current, request v-sync
glfwInit();
GLFWwindow* window = glfwCreateWindow(1200, 900, "Demo", NULL, NULL);
glfwMakeContextCurrent(window);
glfwSwapInterval(1);

// initialize visualization data structures
mjv_defaultCamera(&cam);
mjv_defaultPerturb(&pert);
mjv_defaultOption(&opt);
mjr_defaultContext(&con);

// create scene and context
mjv_makeScene(m, &scn, 1000);
mjr_makeContext(m, &con, mjFONTSCALE_100);

// ... install GLFW keyboard and mouse callbacks

// run main loop, target real-time simulation and 60 fps rendering
while( !glfwWindowShouldClose(window) ) {
  // advance interactive simulation for 1/60 sec
  //  Assuming MuJoCo can simulate faster than real-time, which it usually can,
  //  this loop will finish on time for the next frame to be rendered at 60 fps.
  //  Otherwise add a cpu timer and exit this loop when it is time to render.
  mjtNum simstart = d->time;
  while( d->time - simstart < 1.0/60.0 )
      mj_step(m, d);

  // get framebuffer viewport
  mjrRect viewport = {0, 0, 0, 0};
  glfwGetFramebufferSize(window, &viewport.width, &viewport.height);

  // update scene and render
  mjv_updateScene(m, d, &opt, NULL, &cam, mjCAT_ALL, &scn);
  mjr_render(viewport, &scn, &con);
```

stable

```
  // swap OpenGL buffers (blocking call due to v-sync)
  glfwSwapBuffers(window);

  // process pending GUI events, call GLFW callbacks
  glfwPollEvents();
}

// close GLFW, free visualization storage
glfwTerminate();
mjv_freeScene(&scn);
mjr_freeContext(&con);

// ... free MuJoCo model and data
```

Abstract visualization and interaction

This stage populates the mjvScene data structure with a list of geometric objects, lights, cameras and everything else needed to produce a 3D rendering. It also provides abstract keyboard and mouse hooks for user interaction.

Cameras

There are two types of camera objects: an abstract camera represented with the stand-alone data structure mjvCamera, and a low-level OpenGL camera represented with the data structure mjvGLCamera which is embedded in mjvScene. When present, the abstract camera is used during scene update to automatically compute the OpenGL camera parameters, which are then used by the OpenGL renderer. Alternatively, the user can bypass the abstract camera mechanism and set the OpenGL camera parameters directly, as discussed in the Virtual Reality section below.

The abstract camera can represent three different camera types as determined by mjvCamera.type. The possible settings are defined by the enum mjtCamera:

mjCAMERA_FREE
: This is the most commonly used abstract camera. It can be freely moved with the mouse. It has a lookat point, distance to the lookat point, azimuth and elevation; twist around the line of sight is not allowed. The function mjv_moveCamera is a mouse hook for controlling all these camera properties interactively with the mouse. When simulate.cc first starts, it uses the free camera.

mjCAMERA_TRACKING
: This is similar to the free camera, except the lookat point is no longer a free parameter but instead is coupled to the MuJoCo body whose id is given by

stable

mjvCamera.trackbodyid. At each update, the lookat point is set to the center of mass of the kinematic subtree rooted at the specified body. There is also some filtering which produces smooth camera motion. The distance, azimuth and elevation are controlled by the user and are not modified automatically. This is useful for tracking a body as it moves around, without turning the camera. To switch from the free to the tracking camera in simulate.cc, hold Ctrl and right-double-click on the body of interest. Press Esc to go back to the free camera.

mjCAMERA_FIXED

This refers to a camera explicitly defined in the model, unlike the free and tracking cameras which only exist in the visualizer and are not defined in the model. The id of the model camera is given by mjvCamera.fixedcamid. This camera is fixed in the sense that the visualizer cannot change its pose or any other parameters. However the simulator computes the camera pose at each time step, and if the camera is attached to a moving body or is in tracking or targeting mode, it will move.

mjCAMERA_USER

This means that the abstract camera is ignored during an update and the low-level OpenGL cameras are not changed. It is equivalent to not specifying an abstract camera at all, i.e., passing a NULL pointer to mjvCamera in the update functions explained below.

The low-level mjvGLCamera is what determines the actual rendering. There are two such cameras embedded in mjvScene, one for each eye. Each has position, forward and up directions. Forward corresponds to the negative Z axis of the camera frame, while up corresponds to the positive Y axis. There is also a frustum in the sense of OpenGL, except we store the average of the left and right frustum edges and then during rendering compute the actual edges from the viewport aspect ratio assuming 1:1 pixel aspect ratio. The distance between the two camera positions corresponds to the inter-pupillary distance (ipd). When the low-level camera parameters are computed automatically from an abstract camera, the ipd as well as vertical field of view (fovy) are taken from `mjModel.vis.global.ipd`/`fovy` for free and tracking cameras, and from the camera-specific `mjModel.cam_ipd/fovy` for cameras defined in the model. When stereoscopic mode is not enabled, as determined by mjvScene.stereo, the camera data for the two eyes are internally averaged during rendering.

Selection

In many applications we need to click on a point and determine the 3D this point/pixel belongs. This is done with the function mjv_select, whi collisions. Ray collisions functionality is engine-level and does not depend on the

stable

visualizer (indeed it is also used to simulate rangefinder sensors independent of visualization), but the select function is implemented in the visualizer because it needs information about the camera and viewport.

The function mjv_select returns the index of the geom at the specified window coordinates, or -1 if there is no geom at those coordinates. The 3D position is also returned. See the code sample simulate.cc for an example of how to use this function. Internally, mjv_select calls the engine-level function mj_ray which in turn calls the per-geom functions mj_rayMesh, mj_rayHfield and mju_rayGeom. The user can implement custom selection mechanisms by calling these functions directly. In a VR application for example, it would make sense to use the hand-held controller as a “laser pointer” that can select objects.

Perturbations

Interactive perturbations have proven very useful in exploring the model dynamics as well as probing closed-loop control systems. The user is free to implement any perturbation mechanism of their choice by setting `mjData.qfrc_applied` or `mjData.xfrc_applied` to suitable forces (in generalized and Cartesian coordinates respectively).

Prior to MuJoCo version 1.40, user code had to maintain a collection of objects in order to implement perturbations. All these objects are now grouped into the data structure mjvPerturb. Its use is illustrated in simulate.cc. The idea is to select a MuJoCo body of interest, and provide a reference pose (i.e., a 3D position and quaternion orientation) for that body. These are stored in mjPerturb.refpos/quat. The function mjv_movePerturb is a mouse hook for controlling the reference pose with the mouse. The function mjv_initPerturb is used to set the reference pose equal to the selected body pose at the onset of perturbation, so as to avoid jumps.

This perturbation object can then be used to move the selected body directly (when the simulation is paused or when the selected body is a mocap body), or to apply forces and torques to the body. This is done with the functions mjv_applyPerturbPose and mjv_applyPerturbForce respectively. The latter function writes the external perturbation force to `mjData.xfrc_applied` for the selected body. However it does not clear `mjData.xfrc_applied` for the remaining bodies, thus it is recommended to clear it in user code, in case the selected body changed and some perturbation force was left over from a previous time step. If there is more than one device that can apply perturbations or user code needs to add perturbations from other sources, the user must implement the necessary logic so that only the desired perturba

stable

in `mjData.xfrc_applied` and any old perturbations are cleared.

In addition to affecting the simulation, the perturbation object is recognized by the abstract visualizer and can be rendered. This is done by adding a visual string to denote the positional difference, and a rotating cube to denote the reference orientation of the selected body. The perturbation forces themselves can also be rendered when the corresponding visualization flag in mjvOption is enabled.

Scene update

Finally, we bring all of the above elements together and explain how mjvScene is updated before being passed to the OpenGL rendering stage. This can be done with a single call to the function mjv_updateScene at each frame. mjvCamera and mjvPerturb are arguments to this function, or they can be NULL pointers in which case the corresponding functionality is disabled. In VR applications the parameters of mjvScene.camera[n], n=0,1 must also be set at each frame; this is done by user code outside mjv_updateScene. The function mjv_updateScene examines mjModel and mjData, constructs all geoms that need to be rendered (according to the specified visualization options), and populates the array mjvScene.geom with mjvGeom objects. Note that mjvGeom is an abstract geom, and is not in one-to-one correspondence with the simulation geoms in mjModel and mjData. In particular, mjvGeom contains the geom pose, scale, shape (primitive or mesh index in mjModel), material properties, textures (index in mjModel), labeling, and everything else needed for specify how rendering should be done. mjvScene also contains up to eight OpenGL lights which are copied from the model, as well as a headlight which is in light position 0 when present.

The above procedure is the most common approach, and it updates the entire scene at each frame. In addition, we provide two functions for finer control. mjv_updateCamera updates only the camera (i.e., maps the abstract mjvCamera to the low-level mjvGLCamera) but does not touch the geoms or lights. This is useful when the user is moving the camera rapidly but the simulation state has not changed - in that case there is no point in re-creating the lists of geoms and lights.

More advanced rendering effects can be achieved by manipulating the list of abstract geoms. For example, the user can add custom geoms at the end of the list. Sometimes it is desirable to render a sequence of simulation states (i.e., a trajectory) and not just the current state. For this purpose, we have provided the function mjv_addGeoms which adds the geoms corresponding to the current simulation state to the list already in mjvScene. It does not change the list of lights, because lighting is additive and having too many lights will make the scene too bright. Importantly, the user can select which geom categories will be added, via a bitmask of enum type mjtCatBit:

stable

mjCAT_STATIC

This selects MuJoCo geoms and sites belonging to the world body (which has body id 0).

mjCAT_DYNAMIC

This selects MuJoCo geoms and sites belonging to bodies other than the world body.

mjCAT_DECOR

This selects decorative elements such as force arrows, automatically-generated skeletons, equivalent inertia boxes, and any other elements that were added by the abstract visualizer and do not correspond to MuJoCo geoms and sites defined in the model.

mjCAT_ALL

This selects all of the above categories.

The main update function mjv_updateScene would normally be called with mjCAT_ALL. It clears the geom list and calls mjv_addGeom to add only the geoms for the current model state. If we want to render a trajectory, we have to be careful to avoid visual clutter. So it makes sense to render one of the frames with mjCAT_ALL (usually the first or the last depending on the use case), and all other frames with mjCAT_DYNAMIC. Since the static/world objects are not moving, rendering them in each frame will only slow down the GPU and create visual aliasing. As for the decor elements, there could be situations where we want to render all of them - for example to visualize the evolution of contact forces over time. In summary, there is plenty of flexibility in how mjvScene is constructed. We have provided automation for the main use cases, but the user can also make programmatic changes as needed.

Virtual reality

In desktop applications it is convenient to use an abstract mjvCamera allowing intuitive mouse control, and then automatically map it to mjvGLCamera used for rendering. In VR applications the situation is very different. In that case the head/eyes of the user as well as the projection surface are being tracked, and therefore have physical presence in the room. If anything can be moved by the user (with a mouse or other input device) it is the position, orientation and scale of the model relative to the room. This is called model transformation, and is represented in mjvScene. The function mjv_moveModel is a mouse hook for controlling this transformation. When using an abstract mjvCamera during update, the model transformation is automatically disabled, by setting the flag mjvScene.enabletransform = 0 rather than clearing the actual paramet the user can switch between VR and desktop camera mode without lo transformation parameters.

stable

Since we have introduced two spaces, namely model space and room space, we need to map between them as well as clarify which spatial quantities are defined with respect to which spatial frame. Everything accessible by the simulator lives in the model space. The room space is only accessible by the visualizer. The only quantities defined in room space are the mjvGLCamera parameters. The functions mjv_room2model, mjv_model2room, mjv_cameraInModel, mjv_cameraInRoom perform the necessary transformations, and are needed for VR applications.

We now outline the procedure for hooking up head tracking to MuJoCo's visualizer in a VR application. A code sample illustrating this will soon be posted. We assume that a tracking device provides in real-time the positions of the two eyes (usually generated by tracking the position and orientation of the head and assuming a user-specific ipd), as well as the forward and up camera directions. We copy these data directly into the two mjvGLCameras, which are in mjvScene.camera[n] where n=0 is the left eye and n=1 is the right eye. Note that the forward direction is normal to the projection surface, and not necessarily aligned with the gaze direction; indeed the gaze direction is unknown (unless we also have an eye-tracking device) and does not affect the rendering.

We must also set the mjvGLCamera frustum. How this is done depends on the nature of the VR system. For head-mounted displays such as the Oculus Rift and HTC Vive, the projection surface moves with the head, and so the frustum is fixed and provided by the SDK. In this case we simply copy it into mjvGLCamera, averaging the left and right edges to compute the frustum_center parameter. Alternatively, the projection surface can be a monitor which is stationary in the room (which is the case in the zSpace system). For such systems we must compute the frustum at each frame, by taking into account the spatial relations between the monitor and the eyes/cameras. This assumes that the monitor is also tracked. The natural approach here is to define the monitor as the center of the room coordinate frame, and track the head relative to it. In the zSpace system this is done by embedding the motion capture cameras in the monitor itself.

Apart from tracking the head and using the correct perspective projection, VR applications typically involve hand-held spatial controllers that must be mapped to the motion of simulated objects or otherwise interact with the simulation. The pose of these controllers is recorded by the motion capture system in room space. The transformation functions we provide (mjv_room2model in particular) can be used to map to model space. Once we have the pose of the controller in model space, we can use a MuJoCo mocap body (defined in the model) to insert the controller in the simulation. This is precisely why mocap bodies were introduced in MuJoCo. Such bodies are treated as fixed from the viewpoint of physics, yet the user move them programmatically at each simulation step. They can interact with the simulation through contacts, or better yet, through soft equality constraints to regular bodies which in turn make contacts. The latter approach is illustrated in the MPL models available on the Forum. It provides effective dynamic filtering and avoids contacts involving bodies that behave as if they are infinitely heavy (which is what a fixed body is). Note that the time-varying positions and orientations of the mocap bodies are stored in `mjData.mocap_pos/quat`, as opposed to storing them in mjModel. This is because mjModel is supposed to remain constant. The fixed mocap body pose stored in mjModel is only used at initialization and reset, when user code has not yet had a chance to update mjData.mocap_pos/quat.

OpenGL Rendering

This stage takes the mjvScene data structure populated in the abstract visualization stage, and renders it. It also provides basic 2d drawing and framebuffer access, so that most applications would not need to call OpenGL directly.

Context and GPU resources

The first step in the rendering process is create the model-specific GPU context mjrContext. This is done by first clearing the data structure with the function mjr_defaultContext, and then calling the function mjr_makeContext. This was already illustrated earlier; the relevant code is:

```
mjModel* m;
mjrContext con;

// clear mjrContext only once before first use
mjr_defaultContext(&con);

// create window with OpenGL context, make it current
GLFWwindow* window = glfwCreateWindow(1200, 900, "Demo", NULL, NULL);
glfwMakeContextCurrent(window);

// ... load MuJoCo model

// make model-specific mjrContext
mjr_makeContext(m, &con, mjFONTSCALE_100);

// ... load another MuJoCo model

// make mjrContext for new model (old context freed automatically)
mjr_makeContext(m, &con, mjFONTSCALE_100);

// free context when done
mjr_freeContext(&con);
```

stable

How is mjrContext related to an OpenGL context? An OpenGL context is what enables the application to talk to the video driver and send rendering commands. It must exist and must be current in the calling thread before mjr_makeContext is called. GLFW and related libraries provide the necessary functions as shown above.

mjrContext is specific to MuJoCo. After creation, it contains references (called "names" in OpenGL) to all the resources that were uploaded to the GPU by mjr_makeContext. These include model-specific resources such as meshes and textures, as well as generic resources such as font bitmaps for the specified font scale, framebuffer objects for shadow mapping and offscreen rendering, and associated renderbuffers. It also contains OpenGL-related options copied from `mjModel.vis`, capabilities of the default window framebuffer that are discovered automatically, and the currently active buffer for rendering; see buffers below. Note that even though MuJoCo uses fixed-function OpenGL, it avoids immediate mode rendering and instead uploads all resources to the GPU upfront. This makes it as efficient as a modern shader, and possibly more efficient, because fixed-function OpenGL is now implemented via internal shaders that have been written by the video driver developers and tuned extensively.

Most of the fields of mjrContext remain constant after the call to mjr_makeContext. The only exception is mjrContext.currentBuffer which changes whenever the active buffer changes. Some of the GPU resources may also change because the user can upload modified resources with the functions mjr_uploadTexture, mjr_uploadMesh, mjr_uploadHField. This can be used to achieve dynamic effects such as inserting a video feed into the rendering, or modulating a terrain map. Such modifications affect the resources residing on the GPU, but their OpenGL names are reused, thus the change is not actually visible in mjrContext.

The user should **never** make changes to mjrContext directly. MuJoCo's renderer assumes that only it can manage mjrContext. In fact this kind of object would normally be opaque and its internal structure would not be exposed to the user. We are exposing it because MuJoCo has an open design and also because users may want to interleave their own OpenGL code with MuJoCo's renderer, in which case they may need read access to some fields of mjrContext. For example in VR applications the user needs to blit from MuJoCo's offscreen buffer to a texture provided by a VR SDK.

When a different MuJoCo model is loaded, mjr_makeContext must be called again. There is also the function mjr_freeContext which frees the GPU resources while preserving the initialization and capabilities flags. This function should be called when the application is about to exit. It is called automatically from within m stable
so you do not need to call it directly when a different model is loaded,
an error to do so. The function mjr_defaultContext must be called once before

rendering starts, to clear the memory allocated for the data structure mjrContext. If you call it after calling mjr_makeContext, it will wipe out any record that GPU resources were allocated without freeing those resources, so don't do that.

Buffers for rendering

In addition to the default window framebuffer, OpenGL can support unlimited framebuffer objects (FBOs) for custom rendering. In MuJoCo we provide systematic support for two framebuffers: the default window framebuffer, and one offscreen framebuffer. They are referred to by the constants in the enum type mjtFramebuffer, namely mjFB_WINDOW and mjFB_OFFSCREEN. At any time, one of these two buffers is active for the purposes of MuJoCo rendering, meaning that all subsequent commands are directed to it. There are two additional framebuffer objects referenced in mjrContext, needed for shadow mapping and resolving multi-sample buffers, but these are used internally and the user should not attempt to access them directly.

The active buffer is set with the function mjr_setBuffer. This sets the value of mjrContext.activeBuffer and configures the OpenGL state accordingly. When mjr_makeContext is called, internally it calls mjr_setBuffer with argument mjFB_WINDOW, so that rendering starts in the window buffer by default. If the specified buffer does not exist, mjr_setBuffer automatically defaults to the other buffer (note that when using headless rendering on Linux, there may be no window framebuffer).

From the perspective of OpenGL, there are important differences between the window framebuffer and offscreen framebuffer, and these differences affect how the MuJoCo user interacts with the renderer. The window framebuffer is created and managed by the operating system and not by OpenGL. As a result, properties such as resolution, double-buffering, quad-buffered stereo, multi-samples, v-sync are set outside OpenGL; this is done by GLFW calls in our code samples. All OpenGL can do is detect these properties; we do this in mjr_makeContext and record the results in the various window capabilities fields of mjrContext. This is why such properties are not part of the MuJoCo model; they are session/software-specific and not model-specific. In contrast, the offscreen framebuffer is managed entirely by OpenGL, and so we can create that buffer with whatever properties we want, namely with the resolution and multi-sample properties specified in `mjModel.vis`.

The user can directly access the pixels in the two buffers. This is done with the functions mjr_readPixels, mjr_drawPixels and mjr_blitBuffer. Read/draw transfer pixels from/to the active buffer to/from the CPU. Blit transfers pixels betweer
on the GPU and is therefore much faster. The direction is from the acti

stable

buffer that is not active. Note that mjr_blitBuffer has source and destination viewports that can have different size, allowing the image to be scaled in the process.

Drawing pixels

The main rendering function is mjr_render. Its arguments are a rectangular viewport for rendering, the mjvScene which was populated by the abstract visualizer, and the mjrContext which was created by mjr_makeContext. The viewport can be the entire active buffer, or part of it for custom effects. A viewport corresponding to the entire buffer can be obtained with the function mjr_maxViewport. Note that while the offscreen buffer size does not change, the window buffer size changes whenever the user resizes or maximizes the window. Therefore user code should not assume fixed viewport size. In the code sample simulate.cc we use a callback which is triggered whenever the window size changes, while in basic.cc we simply check the window size every time we render. On certain scaled displays (only on OSX it seems) the window size and framebuffer size can be different. So if you are getting the size with GLFW functions, use glfwGetFramebuferSize rather than glfwGetWindowSize. On the other hand, mouse coordinates are returned by the operating system in window rather than framebuffer units; thus the mouse interaction functions discussed earlier should use glfwGetWindowSize to obtain the window height needed to normalize the mouse displacement data.

mjr_render renders all mjvGeoms from the list mjvScene.geom. The abstract visualization options mjvOption are no longer relevant here; they are used by mjv_updateScene to determine which geoms to add, and as far as mjr_render is concerned these options are already baked-in. There is however another set of rendering options that are embedded in mjvScene, and these affect the OpenGL rendering process. The array mjvScene.flags contains flags indexed by the enum type mjtRndFlag and include options for enabling and disabling wireframe mode, shadows, reflections, skyboxes and fog. Shadows and reflections involve additional rendering passes. MuJoCo's renderer is very efficient, but depending on the model complexity and available GPU, it may be necessary to disable one or both of these effects in some cases.

The parameter mjvScene.stereo determines the stereo mode. The possible values are given by the enum type mjtStereo and are as follows:

mjSTEREO_NONE

Stereo rendering is disabled. The average of the two OpenGL cameras in mjvScene is used. Note that the renderer always expects both ca stable properly defined, even if stereo is not used.

mjSTEREO_QUADBUFFERED

This mode works only when the active buffer is the window, and the window supports quad-buffered OpenGL. This requires a professional video card. The code sample simulate.cc attempts to open such a window. In this mode MuJoCo's renderer uses the GL_BACK_LEFT and GL_BACK_RIGHT buffers to render the two views (as determined by the two OpenGL cameras in mjvScene) when the window is double-buffered, and GL_FRONT_LEFT and GL_FRONT_RIGHT otherwise. If the window does not support quad-buffered OpenGL or the active buffer is the offscreen buffer, the renderer reverts to the side-by-side mode described next.

mjSTEREO_SIDEBYSIDE

This stereo mode does not require special hardware and is always available. The viewport given to mjr_render is split in two equal rectangles side-by-side. The left view is rendered on the left side and the right view on the right side. In principle users can cross their eyes and see stereo on a regular monitor, but the goal here is to show it in a stereoscopic device. Most head-mounted displays support this stereo mode.

In addition to the main mjr_render function, we provide several functions for "decorating" the image. These are 2d rendering functions and include mjr_overlay, mjr_text, mjr_rectangle, mjr_figure. The user can draw additional decorations with their own OpenGL code. This should be done after mjr_render, because mjr_render clears the viewport.

We also provide the functions mjr_finish and mjr_getError for explicit synchronization with the GPU and for OpenGL error checking. They simply call glFinish and glGetError internally. This together with the basic 2d drawing functions above is meant to provide enough functionality so that most users will not need to write OpenGL code. Of course we cannot achieve this in all cases, short of providing wrappers for all of OpenGL.

stable