Physics Simulation

Creating Actors

An actor is an instance of a GymAsset. The function `create_actor` adds an actor to an environment and returns an actor handle that can be used to interact with that actor later. For performance reasons, it is a good practice to save the handles during actor creation rather than looking them up every time while the simulation is running. Many of the accompanying examples do something like this:

```
# cache useful handles
envs = []
actor_handles = []

print("Creating %d environments" % num_envs)
for i in range(num_envs):
    # create env
    env = gym.create_env(sim, env_lower, env_upper, num_per_row)
    envs.append(env)

    # add actor
    actor_handle = gym.create_actor(env, asset, pose, "actor", i, 1)
    actor_handles.append(actor_handle)
```

Actor handles are specific to the environment that the actor was created in. API functions that operate on actors require both the environment reference and actor handle, so those are commonly cached together.

There are quite a few functions that operate on actors. They are named `get_actor_*`, `set_actor_*`, or `apply_actor_*`. Use the API reference for a complete listing.

Aggregates

Note

Aggregates are available only with the PhysX backend. Creating aggregates will have no effect with Flex.

An aggregate is a collection of actors. Aggregates do not provide extra simulation functionality, but allow you to tell PhysX that a set of actors will be clustered together, which in turn allows PhysX to optimize its spatial data operations. It is not necessary to create aggregates, but doing so can provide a modest performance boost. To place multiple actors into an aggregate, you should enclose the calls to `create_actor` in between calls to `begin_aggregate` and `end_aggregate`, like this:

```
gym.begin_aggregate(env, max_bodies, max_shapes, True)
gym.create_actor(env, ...)
gym.create_actor(env, ...)
gym.create_actor(env, ...)
...
gym.end_aggregate(env)
```

Only actors from the same env can be included in an aggregate. When creating an aggregate, it is necessary to specify the maximum number of rigid bodies and shapes, which should be the total number of bodies and shapes in all the actors that will get placed in the aggregate. This information can be obtained from the assets used to create the actors (`get_asset_rigid_body_count` and `get_asset_rigid_shape_count`).

For an example of using aggregates, take a look at `python/rlgpu/tasks/franka.py`.

Actor Components

Each actor has an array of rigid bodies, joints, and DOFs. You can get the counts like this:

```
num_bodies = gym.get_actor_rigid_body_count(env, actor_handle)
num_joints = gym.get_actor_joint_count(env, actor_handle)
num_dofs = gym.get_actor_dof_count(env, actor_handle)
```

It is not possible, at this time, to add or remove actor components once an actor gets created.

Rigid Bodies

Each rigid body constists of one or more rigid shapes. You can customize rigid body and shape properties per actor, as shown in the `body_physics_props.py` example.

Joints

The Gym importers preserves the joints as defined in URDF and MJCF models. Fixed, revolute, and prismatic joints are well-tested and fully supported. Using these joint types, a wide variety of models can be created. Additional support for spherical and planar joints is planned.

Degrees-of-Freedom

Each degree of freedom can be independently actuated. You can apply controls to individual DOFs or use arrays to work with all the actor DOFs at once, as described in the next section.

Controlling Actors

Controlling actors is done using the degrees-of-freedom. For each DOF, you can set the drive mode, limits, stiffness, damping, and targets. You can set these values per actor and override the default settings loaded from the asset.

Scaling Actors

Actors can have their size scaled at runtime. Scaling an actor will change its collision geometry, mass properties, joint positions, and prismatic joint limits. See `examples/actor_scaling.py` for sample usage.

> **Note**
>
> Actor scaling has known limitations when using the GPU pipeline. The center-of-mass of rigid bodies is not updated, which only affects bodies with center-of-mass not at the body origin. Also, resetting transforms and velocities or applying forces in the same simulation step as scaling may yield incorrect results. We expect these limitation to be resolved in the future.

After scaling an actor, some of its properties may have to be manually tuned to get the desired results. For example, its DOF properties (stiffness, damping, max force, armature, etc.), actuation forces, tendon properties etc. may have to be adjusted.

DOF Properties and Drive Modes

DOF property arrays can be accessed for assets (`get_asset_dof_properties`) and individual actors (`get_actor_dof_properties`/`set_actor_dof_properties`). The getters return structured Numpy arrays with the following fields:

Name	Data type	Description
hasLimits	bool	Whether the DOF has limits or has unlimited motion.
lower	float32	Lower limit.
upper	float32	Upper limit.
driveMode	gymapi.DofDriveMode	DOF drive mode, see below.
stiffness	float32	Drive stiffness.
damping	float32	Drive damping.
velocity	float32	Maximum velocity.
effort	float32	Maximum effort (force or torque).
friction	float32	DOF friction.
armature	float32	DOF armature.

`DOF_MODE_NONE` lets the joints move freely within their range of motion:

```
props = gym.get_actor_dof_properties(env, actor_handle)
props["driveMode"].fill(gymapi.DOF_MODE_NONE)
props["stiffness"].fill(0.0)
props["damping"].fill(0.0)
gym.set_actor_dof_properties(env, actor_handle, props)
```

This is used in the `projectiles.py` example, where the ant actors are essentially ragdolls.

`DOF_MODE_EFFORT` lets you apply efforts to the DOF using `apply_actor_dof_efforts`. If the DOF is linear, the effort is a force in Newtons. If the DOF is angular, the effort is a torque in Nm. The DOF properties need to be set only once, but efforts must be applied every frame. Applying efforts multiple times during each frame is cumulative, but they are reset to zero at the beginning of the next frame:

```
# configure the joints for effort control mode (once)
props = gym.get_actor_dof_properties(env, actor_handle)
props["driveMode"].fill(gymapi.DOF_MODE_EFFORT)
props["stiffness"].fill(0.0)
props["damping"].fill(0.0)
gym.set_actor_dof_properties(env, actor_handle, props)

# apply efforts (every frame)
efforts = np.full(num_dofs, 100.0).astype(np.float32)
gym.apply_actor_dof_efforts(env, actor_handle, efforts)
```

`DOF_MODE_POS` and `DOF_MODE_VEL` engage a PD controller that can be tuned using stiffness and damping parameters. The controller will apply DOF forces that are proportional to `posError * stiffness + velError * damping`.

`DOF_MODE_POS` is used for position control, typically with both stiffness and damping set to non-zero values:

```
props = gym.get_actor_dof_properties(env, actor_handle)
props["driveMode"].fill(gymapi.DOF_MODE_POS)
props["stiffness"].fill(1000.0)
props["damping"].fill(200.0)
gym.set_actor_dof_properties(env, actor_handle, props)
```

You can set the position targets using `set_actor_dof_position_targets`:

```
targets = np.zeros(num_dofs).astype('f')
gym.set_actor_dof_position_targets(env, actor_handle, targets)
```

If the DOF is linear, the target is in meters. If the DOF is angular, the target is in radians.

Here’s a more interesting example that sets random positions within the limits defined for each DOF:

```
dof_props = gym.get_actor_dof_properties(envs, actor_handles)
lower_limits = dof_props['lower']
upper_limits = dof_props['upper']
ranges = upper_limits - lower_limits

pos_targets = lower_limits + ranges * np.random.random(num_dofs).astype('f')
gym.set_actor_dof_position_targets(env, actor_handle, pos_targets)
```

`DOF_MODE_VEL` is used for velocity control. The stiffness parameter should be set to zero. The torques applied by the PD controller will be proportional to the damping parameter:

```
props = gym.get_actor_dof_properties(env, actor_handle)
props["driveMode"].fill(gymapi.DOF_MODE_VEL)
props["stiffness"].fill(0.0)
props["damping"].fill(600.0)
gym.set_actor_dof_properties(env, actor_handle, props)
```

You can set the velocity targets using `set_actor_dof_velocity_targets`. If the DOF is linear, the target is in meters per second. If the DOF is angular, the target is in radians per second:

```
vel_targets = np.random.uniform(-math.pi, math.pi, num_dofs).astype('f')
gym.set_actor_dof_velocity_targets(env, actor_handle, vel_targets)
```

Unlike efforts, position and velocity targets don't need to be set every frame, only when changing targets.

Note

When applying controls using `apply_actor_dof_efforts`, `set_actor_dof_position_targets`, and `set_actor_dof_velocity_targets`, you must always pass arrays of length `num_dofs`. A control value from the array will only be applied to its corresponding DOF if that DOF was configured to use that drive mode. For example, if you call `set_actor_dof_position_targets` but one of the DOFs was configured to use `DOF_MODE_EFFORT`, then that DOF will remain in effort mode and its position target value will be ignored.

In addition to working with actor DOF arrays, each DOF can be controlled independently, as shown in the `dof_controls.py` example. Working with individual DOFs is more flexible, but can be less efficient due to the overhead of multiple API calls from Python.

Tensor Control API

The new tensor API provides alternative ways of applying controls. The API for setting up DOF properties remains the same, but you can apply forces or set PD targets using CPU or GPU tensors. This makes it possible to run simulations entirely on the GPU, without copying data between the host and the device.

Physics State

Gym provides an API for getting and setting physics state as structured Numpy arrays.

Rigid Body States

Rigid body states includes position (Vec3), orientation (Quat), linear velocity (Vec3), and angular velocity (Vec3). This allows you to work with the simulation state in maximal coordinates.

You can get rigid body state arrays for an actor, an env, or the entire simulation:

```
body_states = gym.get_actor_rigid_body_states(env, actor_handle, gymapi.STATE_ALL)
body_states = gym.get_env_rigid_body_states(env, gymapi.STATE_ALL)
body_states = gym.get_sim_rigid_body_states(sim, gymapi.STATE_ALL)
```

The methods return structured numpy arrays. The last argument is a bit field that specify the type of state that should be returned. `STATE_POS` means that positions should be computed, `STATE_VEL` means that velocities should be computed, and `STATE_ALL` means that both should be computed. The structure of the returned array is always the same, but the position and velocity values will be computed only if the corresponding flag was set. Internally, Gym maintains a state cache buffer where these values get stored. The Numpy arrays are just wrappers over slices of that buffer. Depending on the underlying physics engine, getting and setting states may require nontrivial computations and the bit flags can be used to avoid unnecessary operations.

The state array slices can be accessed like this:

```
body_states["pose"]             # all poses (position and orientation)
body_states["pose"]["p"])           # all positions (Vec3: x, y, z)
body_states["pose"]["r"])           # all orientations (Quat: x, y, z, w)
body_states["vel"]              # all velocities (linear and angular)
body_states["vel"]["linear"]    # all linear velocities (Vec3: x, y, z)
body_states["vel"]["angular"]   # all angular velocities (Vec3: x, y, z)
```

You can edit and then set the states using the corresponding methods:

```
gym.set_actor_rigid_body_states(env, actor_handle, body_states, gymapi.STATE_ALL)
gym.set_env_rigid_body_states(env, body_states, gymapi.STATE_ALL)
gym.set_sim_rigid_body_states(sim, body_states, gymapi.STATE_ALL)
```

Saving and restoring states is demonstrated in the `projectiles.py` example. At the beginning, we save a copy of the rigid body states for the entire simulation. When the user presses R, those states are restored.

To determine the offset of a specific rigid body in the state arrays, use the `find_actor_rigid_body_index` method, like this:

```
i1 = gym.find_actor_rigid_body_index(env, actor_handle, "body_name", gymapi.DOMAIN_ACTOR)
i2 = gym.find_actor_rigid_body_index(env, actor_handle, "body_name", gymapi.DOMAIN_ENV)
i3 = gym.find_actor_rigid_body_index(env, actor_handle, "body_name", gymapi.DOMAIN_SIM)
```

The last argument specifies the index domain.

- Use domain `DOMAIN_ACTOR` to get an index into the state buffer returned by `get_actor_rigid_body_states`
- Use domain `DOMAIN_ENV` to get an index into the state buffer returned by `get_env_rigid_body_states`
- Use domain `DOMAIN_SIM` to get an index into the state buffer returned by `get_sim_rigid_body_states`

DOF States

You can also work with actors using reduced coordinates:

```
dof_states = gym.get_actor_dof_states(env, actor_handle, gymapi.STATE_ALL)

gym.set_actor_dof_states(env, actor_handle, dof_states, gymapi.STATE_ALL)
```

DOF state arrays include the position and velocity as single floats. For linear DOFs, the positions are in meters and the velocities are in meters per second. For angular DOFs, the positions are in radians and the velocities are in radians per second.

You can access the position and velocity slices like this:

```
dof_states["pos"]   # all positions
dof_states["vel"]   # all velocities
```

You can determine the offset of a specific DOF using the `find_actor_dof_index` method.

Note that DOF states do not include the pose or velocity of the root rigid body, so they don’t fully capture the actor state. As a consequence, we do not provide methods to get and set DOF states for entire environments or simulations. This limitation should be addressed in the future.

The `joint_monkey.py` example illustrates working with DOF states.

Physics State Tensor API

The new tensor API allows for getting and setting state using CPU or GPU tensors. This makes it possible to run simulations entirely on the GPU, without copying data between the host and the device.