Tensor API

The Gym tensor API uses GPU-compatible data representations for interacting with simulations. It allows accessing the physics state directly on the GPU without copying data back and forth from the host. It also supports applying controls using tensors, which makes it possible to set up experiments that run fully on the GPU.

Tensors are essentially multi-dimensional arrays of values. The Gym tensor API works with “global” tensors, i.e., tensors that hold the values for all actors in the simulation. For example, a single tensor is used to hold the state of all the rigid bodies in the simulation. Similarly, a single tensor can be used to apply controls to all actors in the simulation. This approach has important performance benefits. You can get the latest state of all actors or apply controls to all actors using a single function call. This eliminates the need for looping over multiple environments and issuing per-actor operations, which can add considerable overhead, especially in languages like Python. Using global buffers also facilitates data parallelism, particularly on the GPU, where multiple threads can easily work with different slices of the buffers in parallel.

Tensors are well-established data structures for storing GPU-compatible data. Popular frameworks like PyTorch and TensorFlow support tensors as a core feature. The Gym tensor API is independent of other frameworks, but it is designed to be easily compatible with them. The Gym tensor API uses simple tensor desciptors, which specify the device, memory address, data type, and shape of a tensor. There is no special API for manipulating the data in the Gym tensors. Instead, the tensor descriptors can be converted to more usable tensor types, like PyTorch tensors, using interop utilities. Once a Gym tensor is “wrapped” in a PyTorch tensor, you can use all of the existing PyTorch utilties to work with the contents of the tensor.

An important aspect of the tensor API is that it can work with CPU and GPU tensors. This makes it possible to run the same experiment on both CPU and GPU with minimal effort.

Simulation Setup

The tensor API is meant to be used during simulation, after all the environments and actors have been created. To set up the simulation, you can use the “classic” API described in Creating a Simulation. To use the tensor API, there are a few additional details to be aware of.

The tensor API is currently available with PhysX only, so you must use `SIM_PHYSX` when creating the simulation.

To use GPU tensors, you must set the `use_gpu_pipeline` flag to True in the `SimParams` used to create the simulation. Also, you should configure PhysX to use the GPU:

```
sim_params = gymapi.SimParams()
...
sim_params.use_gpu_pipeline = True
sim_params.physx.use_gpu = True

sim = gym.create_sim(compute_device_id, graphics_device_id, gymapi.SIM_PHYSX, sim_params)
```

If `use_gpu_pipeline` is False, the tensors returned by Gym will reside on the CPU.

With the CPU pipeline, PhysX simulation can run on either CPU or GPU, as specified by the `physx.use_gpu` parameter. With the GPU pipeline, PhysX simulation must run on GPU. Because of this, when `use_gpu_pipeline` is True, the `physx.use_gpu` parameter is ignored, and GPU PhysX is used for simulation.

Finally, after all the environments are fully set up, you must call `prepare_sim` to initialize the internal data structures used by the tensor API:

```
gym.prepare_sim(sim)
```

Physics State

After calling `prepare_sim`, you can acquire the physics state tensors. These tensors represent a cache of the simulation state in an easy-to-use format. It is important to note that these tensors hold a copy of the simulation state. They are not the same data structures as used by the underlying physics engine. Each physics engine uses its own data representation and the tensors exposed by Gym are a generic representation that abstracts the underlying details.

Even though the tensors are a copy of the physics state, the process of copying is very fast. When using the GPU pipeline, the data is never copied to the host. Device-to-device copies are fast, so the overhead is minimal. Furthermore, users can control when to refresh the tensors, so no implicit copying is ever done.

Actor Root State Tensor

A Gym actor can consists of one or more rigid bodies. All actors have a root body. The root state tensor holds the state of all the actor root bodies in the simulation. The state of each root body is represented using 13 floats with the same layout as `GymRigidBodyState`: 3 floats for position, 4 floats for quaternion, 3 floats for linear velocity, and 3 floats for angular velocity.

To acquire the root state tensor:

```
_root_tensor = gym.acquire_actor_root_state_tensor(sim)
```

This returns a generic tensor descriptor that isn't very useful by itself, which is why we prefixed the name with an underscore. In order to access the contents of the tensor, you can wrap it in a PyTorch Tensor object, using the provided `gymtorch` interop module:

```
root_tensor = gymtorch.wrap_tensor(_root_tensor)
```

Now you can use this tensor in PyTorch, with all the powerful utilities it provides - no need for us to re-invent all the wheels. In the future, we may add other interop modules, which will allow accessing Gym state using other frameworks like TensorFlow.

The shape of this tensor is (num_actors, 13) and the data type is float32. You can use this tensor as-is, or you can create more convenient views or slices of the data using standard pytorch syntax:

```
root_positions = root_tensor[:, 0:3]
root_orientations = root_tensor[:, 3:7]
root_linvels = root_tensor[:, 7:10]
root_angvels = root_tensor[:, 10:13]
```

You only need to acquire the tensors and create the views once, before the simulation begins.

To update the contents of these tensors with the latest state, call:

```
gym.refresh_actor_root_state_tensor(sim)
```

This function will fill the tensor with the latest values from the physics engine. All the views or slices you created from this tensor will update automatically, since they all refer to the same memory buffer. Generally, you'll want to do this after each call to `gym.simulate`.

Putting it all together, the code might look something like this:

```
# ...create sim, envs, and actors here...

gym.prepare_sim()

# acquire root state tensor descriptor
_root_tensor = gym.acquire_actor_root_state_tensor(sim)

# wrap it in a PyTorch Tensor and create convenient views
root_tensor = gymtorch.wrap_tensor(_root_tensor)
root_positions = root_tensor[:, 0:3]
root_orientations = root_tensor[:, 3:7]
root_linvels = root_tensor[:, 7:10]
root_angvels = root_tensor[:, 10:13]

# main simulation loop
while True:
    # step the physics simulation
    gym.simulate(sim)

    # refresh the state tensors
    gym.refresh_actor_root_state_tensor(sim)

    # ...use the latest state tensors here...
```

You can use the root state tensor to teleport actors by setting new positions, orientations, and velocities for their root bodies. This functionality is useful during resets, but should not be done every frame, because that would interfere with the role of the physics engine and likely lead to non-physical behavior.

As a contrived example, suppose you want to raise all the actors by one unit along the y-axis. You could modify the root positions like this:

```
offsets = torch.tensor([0, 1, 0]).repeat(num_actors)
root_positions += offsets
```

This will modify the root state tensor in-place, and you can apply the changes like this:

```
gym.set_actor_root_state_tensor(sim, _root_tensor)
```

Notice that we are using `_root_tensor` here, which is the original Gym tensor descriptor we acquired at the beginning. We modified the contents of that tensor in-place, and now we are using it to apply an update in the physics engine. The effect is that all actors will teleport one unit upwards, regardless of whether they were single- or multi-body actors.

Another example is doing a periodic reset of actor roots, which would teleport them to their original locations once every 100 steps:

```
# acquire root state tensor descriptor
_root_tensor = gym.acquire_actor_root_state_tensor(sim)

# wrap it in a PyTorch Tensor
root_tensor = gymtorch.wrap_tensor(_root_tensor)

# save a copy of the original root states
saved_root_tensor = root_tensor.clone()

step = 0

# main simulation loop
while True:
    # step the physics simulation
    gym.simulate(sim)

    step += 1

    if step % 100 == 0:
        gym.set_actor_root_state_tensor(sim, gymtorch.unwrap_tensor(saved_root_tensor))
```

Note that we used the torch `clone` method to create a new torch tensor with a copy of the original root states. The Gym method `set_actor_root_state_tensor` cannot digest a torch tensor directly, so we need to convert that torch tensor to a Gym tensor descriptor using `gymtorch.unwrap_tensor`.

To update the state of a subset of actors, there is a method called `set_actor_root_state_tensor_indexed`. It takes an additional tensor of actor indices to reset, which must be 32-bit integers. The total number of actors in the simulation, `A`, can be obtained by calling `gym.get_sim_actor_count(sim)`. Valid actor indices range from 0 to `A` - 1. The index of a specific actor in the root state tensor can be obtained by calling `gym.get_actor_index(env, actor_handle, gymapi.DOMAIN_SIM)`. Here's an example of using `set_actor_root_state_tensor_indexed` with a PyTorch index tensor:

```
actor_indices = torch.tensor([0, 17, 42], dtype=torch.int32, device="cuda:0")

gym.set_actor_root_state_tensor_indexed(sim, _root_states,
gymtorch.unwrap_tensor(actor_indices), 3)
```

Note that the state buffer passed to this method is the same as that passed to `set_actor_root_state_tensor`. In other words, it is the entire tensor containing the states of all actors, but state updates will only be applied to the actors listed in the index tensor. This approach makes life easier in practice and has some performance benefits. You do not need to construct a new state tensor that contains only the states to be modified. You can simply write updated states to the original root state tensor at the correct indices, then specify those indices in a call to `set_actor_root_state_tensor_indexed`.

To avoid some common errors when sharing PyTorch tensors with Gym (and particularly index tensors), please read the Tensor Lifetime section.

Degrees-of-Freedom

Articulated actors have a number of degrees-of-freedom (DOFs) whose state can be queried and changed. The state of each DOF is represented using two 32-bit floats, the DOF position and DOF velocity. For prismatic (translation) DOFs, the position is in meters and the velocity is in meters per second. For revolute (rotation) DOFs, the position is in radians and the velocity is in radians per second.

The DOF state tensor contains the state of all DOFs in the simulation. The shape of the tensor is (num_dofs, 2). The total number of DOFs can be obtained by calling `gym.get_sim_dof_count(sim)`. The DOF states are laid out sequentially. The tensor begins with all the DOFs of actor 0, followed by all the DOFs of actor 1, and so on. The ordering of DOFs for each actor is the same as with the functions `get_actor_dof_states` and `set_actor_dof_states`. The number of DOFs for an actor can be obtained using `gym.get_actor_dof_count(env, actor)`. The global index of a specific DOF in the tensor can be obtained in a number of ways:

- Calling `gym.get_actor_dof_index(env, actor_handle, i, gymapi.DOMAIN_SIM)` will return the global DOF index of the ith DOF of the specified actor.
- Calling `gym.find_actor_dof_index(env, actor_handle, dof_name, gymapi.DOMAIN_SIM)` will look up the global DOF index by name.

The function `acquire_dof_state_tensor` returns a Gym tensor descriptor, which can be wrapped as a PyTorch tensor as previously discussed:

```
_dof_states = gym.acquire_dof_state_tensor(sim)
dof_states = gymtorch.wrap_tensor(_dof_states)
```

The function `refresh_dof_state_tensor` populates the tensor with the latest data from the simulation:

```
gym.refresh_dof_state_tensor(sim)
```

You can modify the values in the DOF state tensor and apply them to the simulation. The function `set_dof_state_tensor` applies all the values in the tensor. This means setting all the DOF positions and velocities for all the actors in the simulation:

```
gym.set_dof_state_tensor(sim, _dof_states)
```

If you modify the values in-place, you can pass the original tensor descriptor obtained from `acquire_dof_state_tensor`. Alternatively, you can create your own tensor to hold the new values, and pass that descriptor instead.

The function `set_dof_state_tensor_indexed` applies the values in the given tensor to the actors specified in the `actor_index_tensor`. The other actors remain unaffected. This is very useful when resetting only selected actors or environments. The actor indices must be 32-bit integers, like those obtained from `get_actor_index`. The following snippet constructs an index tensor and passes it to `set_dof_state_tensor_indexed`:

```
actor_indices = torch.tensor([0, 17, 42], dtype=torch.int32, device="cuda:0")

gym.set_dof_state_tensor_indexed(sim, _dof_states, gymtorch.unwrap_tensor(actor_indices),
3)
```

The state of an articulated actor can be fully defined using its entry in the root state tensor and its entries in the DOF state tensor. Resetting actors whose base is fixed, like manipulator arms, can be accomplished by setting only new DOF states. Resetting actors that are not fixed in place, like walking robots, requires setting a new root state along with new DOF states. In this case, you can use the same actor index tensor, like this:

```
actor_indices = torch.tensor([0, 17, 42], dtype=torch.int32, device="cuda:0")
_actor_indices = gymtorch.unwrap_tensor(actor_indices)

gym.set_actor_root_state_tensor_indexed(sim, _root_states, _actor_indices, 3)
gym.set_dof_state_tensor_indexed(sim, _dof_states, _actor_indices, 3)
```

All Rigid Body States

The rigid body state tensor contains the state of all rigid bodies in the simulation. The state of each rigid body is the same as described for the root state tensor - 13 floats capturing the position, orientation, linear velocity, and angular velocity. The shape of the rigid body state tensor is (num_rigid_bodies, 13). The total number of rigid bodies in a simulation can be obtained by calling `gym.get_sim_rigid_body_count(sim)`. The rigid body states are laid out sequentially.

The tensor begins with all the bodies of actor 0, followed by all the bodies of actor 1, and so on. The ordering of bodies for each actor is the same as with the functions `get_actor_rigid_body_states` and `set_actor_rigid_body_states`. The number of rigid bodies for an actor can be obtained using `gym.get_actor_rigid_body_count(env, actor)`. The global index of a specific rigid body in the tensor can be obtained in a number of ways:

- Calling `gym.get_actor_rigid_body_index(env, actor_handle, i, gymapi.DOMAIN_SIM)` will return the global rigid body index of the ith rigid body of the specified actor.
- Calling `gym.find_actor_rigid_body_index(env, actor_handle, rb_name, gymapi.DOMAIN_SIM)` will look up the global rigid body index by name.

The function `acquire_rigid_body_state_tensor` returns a Gym tensor descriptor, which can be wrapped as a PyTorch tensor as previously discussed:

```
_rb_states = gym.acquire_rigid_body_state_tensor(sim)
rb_states = gymtorch.wrap_tensor(_rb_states)
```

The function `refresh_rigid_body_state_tensor` populates the tensor with the latest data from the simulation:

```
gym.refresh_rigid_body_state_tensor(sim)
```

Presently, the rigid body state tensor is read-only. Setting rigid body states is only allowed for actor root bodies using the root state tensor.

Jacobians and Mass Matrices

The Jacobian and mass matrices are important tools in robotic control. The Jacobian matrix maps the joint space velocities of a DOF to it's cartesian and angular velocities, while the mass matrix contains the generalized mass of the robot depending on the current configuration. They are used in standard robotic control algorithms such as Inverse Kinematics or Operational Space Control.

Both Jacobians and mass matrices are exposed using tensors. The approach is slightly different than other tensor functions, because the size of Jacobian and mass matrices can vary by actor type. When acquiring a Jacobian or mass matrix tensor, you must specify the actor name. The tensor will then contain the matrices for all the actors with that name across all environments - assuming that each actor with that name is of the same type. The actor name is provided when calling `create_actor`. The rest of this section assumes that there is an actor named "franka" in every env that is an instance of the Franka asset:

```
# acquire the jacobian and mass matrix tensors for all actors named "franka"
_jacobian = gym.acquire_jacobian_tensor(sim, "franka")
_massmatrix = gym.acquire_mass_matrix_tensor(sim, "franka")

# wrap as pytorch tensors
jacobian = gymtorch.wrap_tensor(_jacobian)
mm = gymtorch.wrap_tensor(_massmatrix)
```

Note

Jacobian matrices are available with the CPU and GPU pipeline, but mass matrices are currently only available on CPU. Support for GPU mass matrices is planned for the future.

To refresh all of the Jacobians and mass matrix tensors that you acquired, you can call these functions:

```
gym.refresh_jacobian_tensors(sim)
gym.refresh_mass_matrix_tensors(sim)
```

The shape of the mass matrix is (num_dofs, num_dofs). You can obtain the number of DOFs by calling `get_asset_dof_count` or `get_actor_dof_count`. The mass matrix tensor will have shape (num_envs, num_dofs, num_dofs). The Franka asset has 9 DOFs, so with 100 envs the shape of the tensor would be (100, 9, 9).

The shape of the Jacobian depends on the number of links, DOFs, and whether the base is fixed or not. The Jacobian relates the motion of articulation links and DOFs. Rows represent links, columns represent DOFs. Each body has 6 rows in the Jacobian representing its linear and angular motion along the three coordinate axes.

If the actor base is free to move (not fixed in place), the shape of the Jacobian tensor will be (num_envs, num_links, 6, num_dofs + 6). The Franka asset has 11 links and 9 DOFs, so with 100 envs the shape of the tensor would be (100, 11, 6, 15). The extra six DOFs correspond to the linear and angular degrees of freedom of the free root link. To retrieve the matrix elements for a particular link and DOF, you would index into the tensor like this: (env_index, link_index, axis_index, dof_index + 6). For example, to look up the entries for the “panda_hand” link and the “panda_joint5” DOF, you would first look up the link and DOF indices in the asset dictionaries:

```
link_dict = gym.get_asset_rigid_body_dict(franka_asset)
dof_dict = gym.get_asset_dof_dict(franka_asset)

link_index = link_dict["panda_hand"]
dof_index = dof_dict["panda_joint5"]
```

Then you could look up the corresponding entries in the Jacobian tensor like this:

```
# for all envs:
jacobian[:, link_index, :, dof_index + 6]

# for a specific env:
jacobian[env_index, link_index, :, dof_index + 6]
```

If the actor base is fixed, the shape of the Jacobian is different. The root link is not movable, so it doesn't have rows in the Jacobian. Also, there are no root degrees of freedom, so those extra 6 columns are absent. The shape of the Jacobian tensor becomes (num_envs, num_links - 1, 6, num_dofs). Converting our previous example to look up entries in the Jacobian tensor with fixed bases:

```
# for all envs:
jacobian[:, link_index - 1, :, dof_index]

# for a specific env:
jacobian[env_index, link_index - 1, :, dof_index]
```

See Franka IK Picking for an example of using the Jacobian tensor for inverse kinematics.

See Franka OSC for an example of using Jacobians and mass matrices for Operational Space Control.

Contact Tensors

The net contact force tensor contains the net contact forces experienced by each rigid body during the last simulation step, with the forces expressed as 3D vectors. It is a read-only tensor with shape (num_rigid_bodies, 3). You can index individual rigid bodies in the same way as the rigid body state tensor.

The function `acquire_net_contact_force_tensor` returns a Gym tensor descriptor, which can be wrapped as a PyTorch tensor as previously discussed:

```
_net_cf = gym.acquire_net_contact_force_tensor(sim)
net_cf = gymtorch.wrap_tensor(_net_cf)
```

The function `refresh_net_contact_force_tensor` populates the tensor with the latest data from the simulation:

```
gym.refresh_net_contact_force_tensor(sim)
```

Note

The values returned by this function are affected by the contact collection mode (`SimParams.physx.contact_collection`) and the number of physics substeps (`SimParams.substeps`). The contact collection mode allows balancing performance and accuracy in contact reporting. If the contact collection mode is `CC_NEVER`, no contacts are ever collected from the physics engine and the reported net contact forces will always be zero. If the contact collection mode is `CC_ALL_SUBSTEPS`, contacts will be collected during all substeps and the reported net contact forces will be aggregated for all substeps. If the contact collection mode is `CC_LAST_SUBSTEP`, contacts will only be collected during the last substep. Note that when running with a single physics substep, `CC_LAST_SUBSTEP` and `CC_ALL_SUBSTEPS` are equivalent.

The default contact collection mode is `CC_ALL_SUBSTEPS`, which yields the most accurate results but is the slowest mode when running with multiple substeps. `CC_LAST_SUBSTEP` is a more performant option that can be used when full contact reporting precision is not required. It is useful for situations with stable contacts that tend to persist over multiple physics substeps. If the application does not require contact information at all, `CC_NEVER` is the most performant mode that ensures no contact collection overhead.

See **`isaacgym.gymapi.ContactCollection`**.

Force Sensors

You can create rigid body and joint force sensors as described here: Force sensors.

Control Tensors

The various state tensors (root state tensor, DOF state tensor, and rigid body state tensor) are useful for getting information about actors and setting new poses and velocities instantaneously. Setting states this way is appropriate during resets, when actors need to return to their original pose or restart a task using new initial conditions. However, setting new states directly using those tensors should be done sparingly. The positions and velocities of actors are managed by the physics engine, which takes into account collisions, external forces, internal constraints, and drives. Setting new positions and velocities directly should generally be avoided during simulation, because it overrides the physics engine and leads to non-physical behavior. To manage actor behavior during simulation, you can apply DOF forces or PD controls using the following API.

DOF Controls

The function `set_dof_actuation_force_tensor` can be used to apply forces for all DOFs in the simulation. For example, here is how you can apply a random force to all DOFs with PyTorch:

```
# get total number of DOFs
num_dofs = gym.get_sim_dof_count(sim)

# generate a PyTorch tensor with a random force for each DOF
actions = 1.0 - 2.0 * torch.rand(num_dofs, dtype=torch.float32, device="cuda:0")

# apply the forces
gym.set_dof_actuation_force_tensor(sim, gymtorch.unwrap_tensor(actions))
```

Note that the actuation forces will only be applied for DOFs whose `driveMode` was set to `gymapi.DOF_MODE_EFFORT` using `set_actor_dof_properties`. For prismatic (translation) DOFs, the forces are in Newtons. For revolute (rotation) DOFs, the forces are in Nm. The ordering of DOFs in the force tensor is the same as in the DOF state tensor.

The function `set_dof_position_target_tensor` can be used to set PD position targets for all DOFs in the simulation. Note that the position targets will only take effect for DOFs whose `driveMode` was set to `gymapi.DOF_MODE_POS` using `set_actor_dof_properties`. For prismatic (translation) DOFs, the position targets are in meters. For revolute (rotation) DOFs, the position targets are in radians. The ordering of DOFs in the position target tensor is the same as in the DOF state tensor.

The function `set_dof_velocity_target_tensor` can be used to set PD velocity targets for all DOFs in the simulation. Note that the velocity targets will only take effect for DOFs whose `driveMode` was set to `gymapi.DOF_MODE_VEL` using `set_actor_dof_properties`. For prismatic (translation) DOFs, the velocity targets are in meters per second. For revolute (rotation) DOFs, the velocity targets are in radians per second. The ordering of DOFs in the velocity target tensor is the same as in the DOF state tensor.

There are indexed variants of the tensor-based control functions, which can be used to control a subset of actors specified in an index tensor, similarly to `set_dof_state_tensor_indexed`. They are called `set_dof_actuation_force_tensor_indexed`, `set_dof_position_target_tensor_indexed`, and `set_dof_velocity_tensor_indexed`.

Body Forces

You can apply forces to rigid bodies using two different functions. The function `apply_rigid_body_force_tensors` can be used to apply forces and/or torques at the center-of-mass of each rigid body in the simulation:

```
# apply both forces and torques
gym.apply_rigid_body_force_tensors(sim, force_tensor, torque_tensor, gymapi.ENV_SPACE)

# apply only forces
gym.apply_rigid_body_force_tensors(sim, force_tensor, None, gymapi.ENV_SPACE)

# apply only torques
gym.apply_rigid_body_force_tensors(sim, None, torque_tensor, gymapi.ENV_SPACE)
```

The last argument is a `CoordinateSpace` enum, which specifies what space the force and torque vectors are in (`LOCAL_SPACE`, `ENV_SPACE` (default), or `GLOBAL_SPACE`.

If you wish to apply forces or torques to only a selected subset of bodies, make sure to set the corresponding tensor entries to zero.

The other function is `apply_rigid_body_force_at_pos_tensors`, which lets you apply forces to bodies at the given positions. This function will automatically compute and apply the required torques when the force positions are not at the center-of-mass:

```
# apply forces at given positions
gym.apply_rigid_body_force_at_pos_tensors(sim, force_tensor, pos_tensor, gymapi.ENV_SPACE)
```

Please take a look at the `apply_forces.py` and `apply_forces_at_pos.py` examples for sample usage.

Common Problems

Tensor Lifetime

Python uses reference counting for garbage collection. Tensors created using PyTorch are subject to garbage collection as well, just like any regular Python objects. The Gym C++ runtime is a shared library that is not connected with the Python interpreter, so it doesn’t participate in the reference counting of objects. When passing tensors to Gym, it is important to ensure that Python will not garbage-collect the objects while they are still in use. This can be a source of tricky memory errors that are difficult to debug.

One common situation where this can occur is when sharing index tensors with Gym. PyTorch uses LongTensors for indexing, but Gym requires 32-bit indices. Thus we need to convert a LongTensor to a 32-bit integer tensor before passing the indices to Gym. Consider this code:

```
indices_torch = torch.LongTensor([0, 17, 42])
indices_gym = gymtorch.unwrap_tensor(indices_torch.to(torch.int32))
set_actor_root_state_tensor_indexed(sim, states, indices_gym, 3)
```

The problem here is quite subtle. The `unwrap_tensor` function gathers information about the PyTorch tensor in a descriptor that is later passed to Gym. For efficiency, it doesn't make a copy of the tensor's data - it simply records the memory address of the data. However, in the above snippet, the tensor is a temporary variable created by the call to `.to(torch.int32)`. After the call to `unwrap_tensor`, Python doesn't see any references to this temporary tensor, so it can garbage-collect it. If that tensor gets garbage-collected before the call to `set_actor_root_state_tensor_indexed`, the results are undefined - a crash or data corruption are possible.

There is a simple solution to avoid such situations. Whenever you call `unwrap_tensor`, make sure that there is a Python reference to that tensor that will prevent it from being garbage-collected immediately. We can rewrite the above code like this:

```
indices_torch = torch.LongTensor([0, 17, 42])
indices_torch32 = indices_torch.to(torch.int32) # <--- this reference will keep the tensor
alive
set_actor_root_state_tensor_indexed(sim, states, gymtorch.unwrap_tensor(indices_torch32),
3)
```

We are planning to handle this issue more gracefully in a future release.

Limitations

The tensor API is currently only supported for the PhysX backend.

There are also some limitations when using the tensor API with the GPU pipeline. These don't apply with the CPU pipeline, but it is important to keep them in mind when writing code meant to run on both CPU and GPU.

A tensor “setter” function, such as `set_actor_root_state_tensor_indexed`, should be called only once per step. In other words, there should only be one call to a specific setter function in between calls to `gym.simulate`. Instead of calling the function multiple times for different sets of actors, combine the actor indices in such a way that they can be updated in a single call. For example, this code:

```
gym.set_actor_root_state_tensor_indexed(sim, root_states, indices1, n1)
gym.set_actor_root_state_tensor_indexed(sim, root_states, indices2, n2)
```

should be expressed as a single call:

```
gym.set_actor_root_state_tensor_indexed(sim, root_states, combined_indices, n_total)
```

where `combined_indices` is the union of `indices1` and `indices2` and `n_total` is the number of `combined_indices`.

Combining state updates into a single call can also improve performance, so it is a good practice to do so.

Another limitation with the GPU pipeline is that tensor “refresh” functions should be called only once per step, before any calls to tensor setter functions. Calling a tensor refresh function after a tensor setter function may return stale data when there is no call to `gym.simulate` in between. Consider this code:

```
gym.set_dof_state_tensor(sim, dof_states)
gym.refresh_rigid_body_state_tensor(sim)
```

The call to `set_dof_state_tensor` sets new DOF states for all the actors. Under the hood, these new values are recorded, but they do not get applied until a subsequent call to `gym.simulate`. Thus calling `refresh_rigid_body_state_tensor` will return stale values until a call to `gym.simulate` is made. The recommended approach is to formulate algorithms in a way that refreshes state tensors only once after each call to `gym.simulate`, and before any calls are made to tensor setter functions.