# Graphics and Camera Sensors

Isaac Gym exposes APIs to control visual aspects of the scene programattically. Additionally, Isaac Gym exposes API to manage views from many cameras and to treat these cameras as sensors on the robot. The following sections describe camera properties, camera sensors, visual property modification, and other topics related to graphics and camera sensors.

## Camera Properties

All cameras in Isaac Gym, including the viewer's camera, can be created with parameters passed in a GymCameraProperties structure. The camera properties structure contains properties such as field of view, resolution, near and far clipping distances, and a few other options. When creating the viewer in the normal way:

```
viewer = gym.create_viewer(sim, gymapi.CameraProperties())
```

A camera properties structure with default values is constructed and used for the viewer camera. The properties can be customized as in the following code:

```
camera_props = gymapi.CameraProperties()
camera_props.horizontal_fov = 75.0
camera_props.width = 1920
camera_props.height = 1080
viewer = gym.create_viewer(sim, camera_props)
```

**Camera Properties:**

- **width** - image width in pixels
- **height** - image height in pixels
- **horizontal_fov** - horizontal field of view in radians. The vertical field of view will be height/width * horizontal_fov
- **near_plane** - distance from the camera point to the near clipping plane in world units (meters)
- **far_plane** - distance from the camera point to the far clipping plane in world units (meters)

- **supersampling_horizontal** - integer multiplier for supersampling. Setting to a value larger than 1 will result in rendering an image of width*supersampling_horizontal pixels wide and resampling to output resolution
- **supersampling_vertical** - integer multiplier for supersampling. Setting to a value larger than 1 will result in rendering an image of height*supersampling_vertical pixels tall and resampling to output resolution
- **use_collision_geometry** - if set to `True`, camera will render collision geometry
- **enable_tensors** - if set to `True`, tensor access will be enabled for this camera. See *GPU Tensor Access* section below for more.

## Camera Sensors

Camera sensors in Isaac Gym are meant to simulate cameras as they would be found either observing or attached to a robot. A camera sensor can be easily created as follows:

```
camera_props = gymapi.CameraProperties()
camera_props.width = 128
camera_props.height = 128
camera_handle = gym.create_camera_sensor(env, camera_props)
```

As with the viewer camera, a customized camera properties can also be passed in to the camera creation. A camera sensor differs from the viewer camera in that it cannot be controled through the viewer and it is connected to a specific environment. This means that when a camera sensor creates an image, it can 'see' only those actors in the selected environment.

The location of camera sensors can be set in one of three ways. First, a camera sensor may be placed directly using:

```
gym.set_camera_location(camera_handle, env, gymapi.Vec3(x,y,z), gymapi.Vec3(tx,ty,yz))
```

where `(x,y,z)` are the coordinates of the camera in the environment's local coordinate frame and `(tx, ty,tz)` are the coordinates of the point the camera is looking at, again in the environments local coordinates.

A second method to place the camera is to directly specify a `GymTransform` which specifies the position and pose of the camera in environemnt local coordinates, such as:

```
transform = gymapi.Transform()
transform.p = (x,y,z)
transform.r = gymapi.Quat.from_axis_angle(gymapi.Vec3(0,1,0), np.radians(45.0))
gym.set_camera_transform(camera_handle, env, transform)
```

Finally, a camera can be directly attached to a rigid body. This is useful to create cameras which track a body as it moves or to simulate the camera being directly attached to a part of an actor. Cameras can be attached to a rigid body as follows:

```
local_transform = gymapi.Transform()
local_transform.p = (x,y,z)
local_transform.r = gymapi.Quat.from_axis_angle(gymapi.Vec3(0,1,0), np.radians(45.0))
gym.attach_camera_to_body(camera_handle, env, body_handle, local_transform,
gymapi.FOLLOW_TRANSFORM)
```

The last argument determines the attachment behavior:

- `gymapi.FOLLOW_POSITION` means that the camera will maintain a fixed offset (`local_transform.p`) from the rigid body, but will not rotate with the body.
- `gymapi.FOLLOW_TRANSFORM` means that the camera will maintain a fixed offset and also rotate with the body.

When attached to a rigid body, the camera's transform will be updated in `step_graphics(sim)`.

All camera sensors are rendered together in a single API call to enable best performance:

```
gym.render_all_camera_sensors(sim)
```

After `render_all_camera_sensors(sim)`, all outputs from all cameras will be ready for retrieval.

## Camera Image Types

Each camera renders a number of different output images. When accessing images, APIs take an image selector which indicates which of the output images the application would like to retrieve or access.

### Camera Sensor Image Types:

- **IMAGE_COLOR** - *4x 8 bit unsigned int* - RGBA color

- **IMAGE_DEPTH** - *float* - negative distance from camera to pixel in view direction in world coordinate units (meters)
- **IMAGE_SEGMENTATION** - *32bit unsigned int* - ground truth semantic segmentation of each pixel. See
- **IMAGE_OPTICAL_FLOW** - *2x 16bit signed int* - screen space motion vector per pixel, normalized

The depth image in linearized back into world coordinates. This means that the value of the pixel is in the units of the simulation, which is meters. Each pixel of the segmentation image contains the segmentation ID of the body which generated the pixel. See the section *Segmentation Image* below. The values returned for the optical flow image are 16 bit signed integers. The following code shows how to convert a value from optical flow image to a value in pixel units:

```
optical_flow_image = gym.get_camera_image(sim, camera_handle, gymapi.IMAGE_OPTICAL_FLOW)
optical_flow_in_pixels = np.zeros(np.shape(optical_flow_image))
# Horizontal (u)
optical_flow_in_pixels[0,0] = image_width*(optical_flow_image[0,0]/2**15)
# Vertical (v)
optical_flow_in_pixels[0,1] = image_height*(optical_flow_image[0,1]/2**15)
```

## Accessing Camera Images

Images rendered with camera sensors can be accessed by calling `get_camera_image` for the desired image type as follows:

```
color_image = gym.get_camera_image(sim, env, camera_handle, gymapi.IMAGE_COLOR)
```

The third argument selects which image from that camera should be returned. The above code asks for the color (RGBA) image. The image is returned as a numpy array. When the image contains vector outputs, such as `IMAGE_COLOR` and `IMAGE_OPTICAL_FLOW`, these vectors are tightly packed in the fastest moving dimension.

See graphics example.

### GPU Tensor Access

When using camera sensor outputs for training a model which is already present on the GPU, a key optimization is to prevent copying the image to the CPU in Isaac Gym only to have the learning framework copy it back to the GPU. To enable better performance, Isaac Gym provides a method for direct GPU access to camera outputs without copying back to CPU memory.

In order to activate tensor access, a camera must be created with the `enable_tensors` flag set to `True` in the camera properties structure.:

```
camera_props = gymapi.CameraProperties()
camera_props.enable_tensors = True
cam_handle = gym.create_camera_sensor(env, camera_props)
```

During the setup phase, the application should retrieve and store the tensor structure for the camera by calling:

```
camera_tensor = gym.get_camera_image_gpu_tensor(sim, env, cam_handle, gymapi.IMAGE_COLOR)
```

The returned GPU tensor has a gpu device side pointer to the data resident on the GPU as well as information about the data type and tensor shape. Sharing this data with a deep learning framework requires a tensor adapter, like the one provided in the `gymtorch` module for PyTorch interop:

```
camera_tensor = gym.get_camera_image_gpu_tensor(sim, env, cam_handle, gymapi.IMAGE_COLOR)
torch_camera_tensor = gymtorch.wrap_tensor(camera_tensor)
```

Now during the main loop of the simulation, the application must declare when it starts and stops accessing the tensors to prevent memory hazards.:

```
while True:

    gym.simulate(sim)
    gym.fetch_results(sim, True)
    gym.step_graphics(sim)
    gym.render_all_camera_sensors(sim)
    gym.start_access_image_tensors(sim)
    #
    # User code to digest tensors
    #
    gym.end_access_image_tensors(sim)
```

The `start_access_image_tensors(sim)` API informs Isaac Gym that the user code wants to access the image tensor buffers directly. This API will stall until all of the image tensors from the previous call to `render_all_camera_sensors(sim)` are written. To minimize stalling, this API is

written to stall once for all of the camera sensors which were created with the `enable_tensors` flag set to `True`.

Isaac Gym will not write to the tensors again until `end_access_image_tensors(sim)` is called. By convention, access to image tensors should be started and stopped within the same iteration of the main loop. Calling `render_all_camera_sensors(sim)` between the start and end of tensor access may lead to an application deadlock.

See PyTorch interop example and the chapter on the tensor API for more information.

# Visual Properties

Isaac Gym contains a number of APIs designed to allow programmatic modification of visual attributes for techniques like visual domain randomization. This section briefly touches on the routines for modifying visual attributes of a body or scene.

## Colors

The color of any rigid body can be set with the API:

```
gym.set_rigid_body_color(env, actor_handle, rigid_body_index,
gymapi.MESH_VISUAL_AND_COLLISION, gymapi.Vec3(r, g, b))
```

In the above call, `env` is the environment containing the actor, `actor_handle` is a handle to the actor in the environment and `rigid_body_index` is the index of the rigid body within the actor whose color is to be modified. The 4th argument is a mesh selector and allows the caller to specify whether to set the color of the rigid body's visual mesh, `MESH_VISUAL`, the rigid body's collision mesh, `MESH_COLLISION`, or both, `MESH_VISUAL_AND_COLLISION`. The last argument is a color vector of the new color where `red`, `green`, and `blue` are floating point values between 0.0 and 1.0. This sets the color for the entire rigid body. If the mesh has submeshes with different colors, the color of all submeshes is set to the specified color.

The current color of a rigid body can also be queried with:

```
current_color = gym.get_rigid_body_color(env, actor_handle, rigid_body_index,
gymapi.MESH_VISUAL)
```

When `get_rigid_body_color` is called with the mesh selector `MESH_VISUAL_AND_COLLISION`, the color of the visual mesh is returned. If the mesh contains submeshes, the color of the first submesh is returned.

## Textures

Isaac Gym provides two routines for creating textures from the API, one that reads a texture from a file and one that accepts a numpy array of pixels to be used as a texture. Textures are created from a file using:

```
texture_handle = gym.create_texture_from_file(sim, texture_filename);
```

Valid file types are .jpg, .png, and .bmp. Textures also can be created from a buffer using:

```
texture_handle = gym.create_texture_from_buffer(sim, width, height, pixelArray)
```

where `pixelArray` is a numpy array of type uint8_t with size [height, width*4] of packed RGBA values. Alpha values should always be 255 on input.

Both methods for creating a texture allocate a texture within the graphics system, copy the pixel data to the GPU, and finally return a handle that can be used in the future to refer to that texture. Since texture creation requires memory allocation and copy to the GPU, it is strongly recommended that all textures needed in a simulation be created during setup and bodies be modified using the handles during simulation.

The texture of a rigid body can be set in much the same way as the color, with the color replaced by a texture handle returned by either `create_texture_from_file()` or `create_texture_from_buffer()` as follows:

```
gym.set_rigid_body_texture(env, actor_handle, rigid_body_index,
gymapi.MESH_VISUAL_AND_COLLISION, texture_handle)
```

Texture, like color, can be queried in the same fashion:

```
texture_handle = gym.get_rigid_body_texture(env, actor_handle, rigid_body_index,
gymapi.MESH_VISUAL)
```

If there is not a texture assigned to the selected rigid body, `INVALID_HANDLE` is returned. Like color, if the mesh selector is `MESH_VISUAL_AND_COLLISION`, the texture assigned to the visual asset, if any, is returned.

Texture coordinates are loaded with the mesh when the asset is created. If a mesh does not contain any texture coordinates, they are generated using a spherical projection at the time the asset is loaded.

If a texture is no longer needed it can be freed using:

```
gym.free_texture(sim, texture_handle)
```

All textures and mesh data are automatically freed from GPU memory when the `GymSim` is destroyed.

### Texture & Color Interaction

Texture and color are multipicatively applied by the renderer. This means that the texture sample at a pixel is multiplied by the body color to calculate the resulting pixel color. For normal application of textures, set the color of the rigid body to `(1.0, 1.0, 1.0)` when setting the texture to guarantee the correct color. However, this feature can be used to increase the space of domain randomization for a small set of textures by tinting the textures or it could be used to tint a body according to the forces it is experiencing while maintaining its texture.

## Reset

At any time the color and texture for an entire actor can be reset to the original value loaded from the asset file by calling:

```
gym.reset_actor_materials(env, actor, gymapi.MESH_VISUAL_AND_COLLISION)
```

This will reset the materials for all bodies in the actor of the specified type: visual, collision, or both.

## Segmentation ID

In order to differentiate different parts of the robot in the semantic segmentation ground truth image (IMAGE_SEGMENTATION), it is necessary to assign differing segmentation ID values to differing bodies. The segmentation ID of a body is set much like color and texture:

```
gym.set_rigid_body_segmentation_id(env, actor_handle, rigid_body_index, segmentation_id)
```

When `IMAGE_SEGMENTATION` is retrieved from a camera, any pixel which originated from the indicated body will have a value of `segmentation_id`, which is a 32-bit unsigned integer.

### Lights

Isaac Gym's rendering has a limited set of lights that can be controlled programatically with the API:

```
gym.set_light_parameters(sim, light_index, intensity, ambient, direction)
```

`light_index` is the index of the light, only values 0 throuhg 3 are valid . Intensity is a Vec3 of the relative RGB values for the light intensity, where 0 is off and 1.0 is max intensity. There is no physical correspondence for the intensity values. `ambient` is a Vec3 of RGB values for the ambient lighting in the same fashion as the classical phong model. Finally, `direction` is a Vec3 which gives the direction of the light. All lights are directional lights only.

The lighting model ise the only part of Isaac Gym's graphics system which is shared globally. Calling `set_light_parameters` affects the RGB images of all camera sensors in all environments and the viewer universally.

## Lines API

The lines API is a simple API for drawing lines on top of the rendered image in the viewer for debugging or visualization purposes. Lines can be used to visualize contacts, applied forces, or any other vector quantity reasonably easily. The lines API is very rudimentary, with one API to add lines to an accumulator of lines, and another API to flush all lines. The API to add lines is as follows:

```
gym.add_lines(viewer, env, num_lines, vertices, colors)
```

In this call, `vertices` is an array of size [num_lines * 2 , 3] where element [2 * i, :] specifies a 3D coordinate in the local frame of env for the start and element 2*i+1 specifies the coordinate of the end of line i. `colors` is an array of size [num_lines, 3] where every index [i,:] is an RGB color for line i.

All of the lines that have been added for drawing are removed with a call to:

```
gym.clear_lines(viewer)
```

A common use for the lines API is to add lines right before drawing the viewer and clearn them immediately afterward:

```
while not gym.query_viewer_has_closed(viewer):
    gym.simulate(sim)
    gym.fetch_results(sim, True)
    gym.step_graphics(sim)

    # generate_lines is a user-written function which creates the desired line vertices
    line_vertices, line_colors, num_lines = generate_lines()

    gym.add_lines(viewer, env, num_lines, line_vertices, line_colors)
    gym.draw_viewer(viewer)
```

Lines drawn with the Lines API are only visible in the viewer. They are not rendered into the view of camera sensors.

## Camera Intrinsics

Some advanced uses of Isaac Gym, such as deprojecting depth images to 3D point clouds, requires complete knowledge of the projection terms used to create the output images. To aid in this, Isaac Gym provides access to the projection and view matrix used to render a camera's view. They are:

```
projection_matrix = np.matrix(gym.get_camera_proj_matrix(sim, env, camera_handle))

view_matrix = np.matrix(gym.get_camera_view_matrix(sim, env, camera_handle))
```

Both of these functions return a numpy array, which is converted to a numpy matrix in the code samples above.

## Step Graphics

When rendering is required, transforms and information must be communicated from the physics simulation into the graphics system. In order to support use cases in which graphics and physics are not running at the same update rate, e.g. if graphics is rendering only every Nth step, Isaac Gym allows manual control over this process. This is further important when many camera sensors are used as the renderer must know when it has moved to a new frame in time, but render() calls are not implicitly frame boundaries since there may be many render() calls per step for viewers and camera sensors. The frame boundary update is made explicit with the `step_graphics(sim)` API.

In most cases, however, `step_graphics()` will be called immediately before a call to `draw_viewer()` and/or `render_all_camera_sensors()` as follows:

```
while not gym.query_viewer_has_closed(viewer):
    gym.simulate(sim)
    gym.fetch_results(sim, True)

    gym.step_graphics(sim)
    gym.render_all_camera_sensors(sim)
    gym.draw_viewer(viewer)
```