pybullet quickstart guide

draft version, check most up-to-date version online

Introduction

pybullet is an easy to use Python module for physics simulation, robotics and machine learning. With pybullet you can load articulated bodies from URDF, SDF and other file formats. pybullet provides forward dynamics simulation, inverse dynamics computation, forward and inverse kinematics and collision detection queries.

Aside from physics simulation, there are bindings to rendering, with a CPU renderer (TinyRenderer) and OpenGL visualization and support for Virtual Reality headsets such as HTC Vive and Oculus Rift. pybullet has also functionality to perform collision detection queries (closest points, overlapping pairs etc) and to add debug rendering (debug lines and text).

We designed the pybullet API to be independent from the underlying physics engine and render engine, so we can easily migrate to newer versions of Bullet, or use a different physics engine or render engine. By default, pybullet uses the Bullet 2.x API on the CPU. We will expose Bullet 3.x running on GPU using OpenCL as well.

pybullet can be easily used with TensorFlow and frameworks such as OpenAI Gym.

Hello pybullet World

Here is a pybullet introduction script that we discuss step by step:

```
import pybullet as p
p.connect(p.DIRECT)
p.setGravity(0,0,-10)
planeId = p.loadURDF("plane.urdf")
cubeStartPos = [0,0,1]
cubeStartOrientation = p.getQuaternionFromEuler([0,0,0])
boxId = p.loadURDF("r2d2.urdf",cubeStartPos, cubeStartOrientation)
p.stepSimulation()
cubePos, cubeOrn = p.getBasePositionAndOrientation(boxId)
print(cubePos,cubeOrn)
```

connect (DIRECT, GUI, SHARED_MEMORY, UDP)

After importing the pybullet module, the first thing to do is 'connecting' to the physics simulation. pybullet is designed around a command-status driven API, with a client sending commands and a server returning the status. pybullet has some build-in physics servers: DIRECT and GUI.

The DIRECT connection sends the commands directly to the physics engine, without using any transport layer, and directly returns the status after executing the command.

The GUI connection will create a new graphical user interface (GUI) with 3D OpenGL rendering, within the same process space as pybullet. On Linux and Windows this GUI runs in a separate thread, while on OSX it runs in the same thread due to operating system limitations. The commands and status messages are send between pybullet client and the GUI physics simulation server using an ordinary memory buffer.

It is also possible to connect to a physics server in a different process on the same machine or on a remote machine using SHARED_MEMORY or UDP networking. See the section about Shared Memory and UDP for details.

connect input arguments

required	connection mode	integer: DIRECT, GUI, SHARED_MEMORY, UDP	DIRECT mode create a new physics engine and directly communicates with it. GUI will create a physics engine with graphical GUI frontend and communicates with it. SHARED_MEMORY will connect to an existing physics engine process on the same machine, and communicates with ot over shared memory. UDP will connect to an existing physics server over UDP networking.
optional	UdpNetworkAddress (UDP mode only)	string	IP address or host name, for example "127.0.0.1" or "mymachine.domain.com"
optional	UdpNetworkPort (UDP mode only)	integer	UDP port number

pybullet connect to a physics server using Shared Memory

There are a few physics servers that allow shared memory connection: the Bullet Example Browser has one example under Experimental/Physics Server that allows shared memory connection.

You can also connect over shared memory to the App_SharedMemoryPhysics_VR, the Virtual Reality application with support for head-mounted display and 6-dof tracked controllers such as

HTC Vive and Oculus Rift with Touch controllers. Since the Valve OpenVR SDK only works properly under Windows, the App_SharedMemoryPhysics_VR can only be build under Windows using premake.

pybullet connect to a physics server using UDP networking

For UDP networking, there is a App_PhysicsServerUDP that listens to a certain UDP port. It uses the open source enet library for UDP networking.

One more UDP application is the App_PhysicsServerSharedMemoryBridgeUDP application that acts as a bridge to an existing physics server: you can connect over UDP to this bridge, and the bridge connects to a physics server using shared memory: the bridge passes messages between client and server.

setGravity

By default, there is no gravitational force enabled. setGravity lets you set the default gravity force for all objects.
setGravity input parameters (no return value)

required	gravityX	float	gravity force along the X world axis
required	gravityY	float	gravity force along the Y world axis
required	gravityZ	float	gravity force along the Z world axis

loadURDF

The loadURDF will send a command to the physics server to load a physics model from a Universal Robot Description File (URDF). The URDF file is used by the ROS project (Robot Operating System) to describe robots and other objects, it was created by the WillowGarage and the Open Source Robotics Foundation (OSRF). Many robots have public URDF files, you can find a description and tutorial here: http://wiki.ros.org/urdf/Tutorials

loadURDF arguments

required	fileName	string	a relative or absolute path to the URDF file on the file system of the physics server.
optional	basePosition	vec3	create the base of the object at the specified position in world space coordinates [X,Y,Z]

optional	baseOrientation	vec4	create the base of the object at the specified orientation as world space quaternion [X,Y,Z,W]
optional	useMaximalCoordinates	int	By default, the joints in the URDF file are created using the reduced coordinate method: the joints are simulated using the Featherstone Articulated Body algorithm (btMultiBody in Bullet 2.x). The useMaximalCoordinates option will create a 6 degree of freedom rigid body for each link, and constraints between those rigid bodies are used to model joints.
optional	useFixedBase	int	

loadURDF returns a body unique id, a non-negative integer value. If the URDF file cannot be loaded, this integer will be negative and not a valid body unique id.

getQuaternionFromEuler and getEulerFromQuaternion

The pybullet API uses quaternions to represent orientations. Since quaternions are not very intuitive for people, there are two APIs to convert between quaternions and Euler angles.
The getQuaternionFromEuler input arguments are:

required	eulerAngle	vec3: list of 3 floats	The X,Y,Z Euler angles are in radians, accumulating 3 rotations around the X, Y and Z axis.

getQuaternionFromEuler returns a list of 3 floating point values, a vec3.

The getEulerFromQuaternion input arguments are:

required	quaternion	vec4: list of 4 floats	The quaternion format is [x,y,z,w]

getEulerFromQuaternion returns a quaternion, vec4 list of 4 floating point values [X,Y,Z,W]

stepSimulation

stepSimulation will perform all the actions in a single forward dynamics simulation step such as collision detection, constraint solving and integration. stepSimulation takes no input arguments and has no return values.

See also setRealTimeSimulation to automatically let the physics server run forward dynamics simulation based on its real-time clock.

getBasePositionAndOrientation

getBasePositionAndOrientation reports the current position and orientation of the base (or root link) of the body in Cartesian world coordinates. The orientation is a quaternion in [x,y,z,w] format.

The getBasePositionAndOrientation input parameters are:

required	objectUniqueId	int	object unique id, as returned from loadURDF.

getBasePositionAndOrientation returns the position list of 3 floats and orientation as list of 4 floats in [x,y,z,w] order. Use getEulerFromQuaternion to convert the quaternion to Euler if needed.

See also resetBasePositionAndOrientation to reset the position and orientation of the object.

This completes the first pybullet script. Bullet ships with several URDF files in the Bullet/data folder.

Controlling a robot

In the Introduction we already showed how to initialize pybullet and load some objects. If you replace the file name in the loadURDF command with "r2d2.urdf" you can simulate a R2D2 robot from the ROS tutorial. Let's control this R2D2 robot to move, look around and control the gripper. For this we need to know how to access its joint motors.

Base, Joints, Links

A simulated robot as described in a URDF file has links connected by joints. Each joint connects a parent link to a child link. At the root of the hierarchy there is a single root parent, that we call base. The base can be either fully fixed, 0 degrees of freedom, or fully free, with 6 degrees of freedom. Each link is connected to a parent with a single joint, so the number of joints is equal to the number of links. Regular links have link indices in the range [0..getNumJoints()] Since the base is not a regular 'link', we use the convention of -1 as its link index.

getNumJoints

After you load a robot you can query the number of joints using the getNumJoints API. For the r2d2.urdf this should return 15.

getNumJoints input parameters:

required	bodyUniqueId	int	the body unique id, as returned by loadURDF etc.

getNumJoints returns an integer value representing the number of joints.

getJointInfo

For each joint we can query some information, such as its name and type.

getJointInfo input parameters

required	bodyUniqueId	int	the body unique id, as returned by loadURDF etc.
required	jointIndex	int	an index in the range [0 .. getNumJoints(bodyUniqueId)]

getJointInfo returns a list of information:

jointIndex	int	the same joint index as the input parameter

jointName	string	the name of the joint, as specified in the URDF (or SDF etc) file
jointType	int	type of the joint, this also implies the number of position and velocity variables. JOINT_REVOLUTE, JOINT_PRISMATIC, JOINT_SPHERICAL, JOINT_PLANAR, JOINT_FIXED. See the section on Base, Joint and Links for more details.
qIndex	int	the first position index in the positional state variables for this body
uIndex	int	the first velocity index in the velocity state variables for this body
flags	int	reserved
jointDamping	float	the joint damping value, as specified in the URDF file
jointFriction	float	the joint friction value, as specified in the URDF file

setJointMotorControl

We can control a robot by setting a desired control mode for one or more joint motors. During the stepSimulation the physics engine will simulate the motors to reach the given target value that can be reached within the maximum motor forces and other constraints. Each revolute joint and prismatic joint is motorized by default. There are 3 different motor control modes: position control, velocity control and torque control.

p.setJointMotorControl(bodyIndex,jointIndex,mode,target,maxforce)

required	bodyUniqueId	int	body unique id as returned from loadURDF etc.
required	linkIndex	int	link index in range [0..getNumJoints(bodyUniqueId)
required	controlMode	int	POSITION_CONTROL, VELOCITY_CONTROL, TORQUE_CONTROL
required	targetValue	float	in POSITION_CONTROL the targetValue is target position of the joint. in VELOCITY_CONTROL the targetValue is target velocity of the joint
required	maxMotorForce	float	in POSITION_CONTROL and VELOCITY_CONTROL this is the maximum motor force used to reach the target value

getJointState

We can query several state variables from the joint using getJointState, such as the joint position, velocity, joint reaction forces and joint motor torque.

getJointState input parameters

required	bodyUniqueId	int	body unique id as returned by loadURDF etc
required	linkIndex	int	link index in range [0..getNumJoints(bodyUniqueId)]

getJointState output

jointPosition	float	The position value of this joint.
jointVelocity	float	The velocity value of this joint.
jointReactionForces	list of 6 floats	There are the joint reaction forces, if a torque sensor is enabled for this joint. Without torque sensor, it is [0,0,0,0,0,0].
appliedJointMotorTorque	float	This is the motor torque applied during the last stepSimulation

getLinkState

You can also query the Cartesian world position and orientation for the center of mass of each link using getLinkState. It will also report the local inertial frame of the center of mass to the URDF link frame, to make it easier to compute the graphics/visualization frame.

getLinkState input parameters

required	bodyUniqueId	int	
required	linkIndex	int	

getLinkState return values

linkWorldPosition	vec3, list of 3 floats	Cartesian position of center of mass
linkWorldOrientation	vec4, list of 4 floats	Cartesian orientation of center of mass, in quaternion [x,y,z,w]
localInertialFramePosition	vec3, list of 3 floats	local position offset of inertial frame (center of mass) to URDF link frame
localInertialFrameOrientation	vec4, list of 4 floats	local orientation (quaternion [x,y,z,w]) offset of the inertial frame to the URDF link frame.

KUKA arm
quadruped robot

Loading models from files

- SDF
- Bullet
- MJCF

Example scripts

examples/pybullet/quadruped.py	load a quadruped from URDF file, step the simulation, control the motors for a simple hopping gait based on sine waves.
examples/pybullet/testrender.py	load a URDF file and render an image, get the pixels (RGB, depth, segmentation mask) and display the image using MatPlotLib.
examples/pybullet/testrender_np.py	Similar to testrender.py, but speed up the pixel transfer using NumPy arrays. Also includes simple benchmark/timings.
examples/pybullet/saveWorld.py	Save the state (position, orientation) of objects into a pybullet Python scripts. This is mainly useful to setup a scene in VR and save the initial stte. Not all state is serialized.
Example pybullet for OpenAI GYM, DeepMind Lab	work-in-progress. See cartpole++ http://matpalm.com/blog/cartpole_plus_plus/ https://github.com/matpalm/cartpoleplusplus

Synthetic Camera Rendering

pybullet has a build-in CPU renderer based on TinyRenderer. This makes it very easy to render images from an arbitrary camera position.

The synthetic camera is specified by two 4 by 4 matrices: the view matrix and the projection matrix. Since those are not very intuitive, there are some helper methods to compute the view and projection matrix from understandable parameters.

computeViewMatrix

The computeViewMatrix input parameters are

required	cameraEyePosition	vec3, list of 3 floats	eye position in Cartesian world coordinates
required	cameraTargetPosition	vec3, list of 3 floats	position of the target (focus) point, in Cartesian world coordinates
required	cameraUpVector	vec3, list of 3 floats	up vector of the camera, in Cartesian world coordinates

Output is the 4x4 view matrix, stored as a list of 16 floats.

computeViewMatrixFromYawPitchRoll

The input parameters are

required	cameraTargetPosition	list of 3 floats	target focus point in Cartesian world coordinates
required	distance	float	distance from eye to focus point
required	yaw	float	yaw angle in degrees, up and down
required	pitch	float	pitch in degrees around up vector
required	roll	float	roll in degrees around forward vector
required	upAxisIndex	int	either 1 for Y or 2 for Z axis up.

Output is the 4x4 view matrix, stored as a list of 16 floats.

computeProjectionMatrix

The input parameters are

required	left	float	
required	right	float	
required	bottom	float	
required	top	float	
required	near	float	
required	far	float	

Output is the 4x4 projection matrix, stored as a list of 16 floats.

computeProjectionMatrixFOV

getCameraImage

The getCameraImage API will return a RGB image, a depth buffer and a segmentation mask buffer with body unique ids of visible objects for each pixel.

getCameraImage input parameters:

required	width	int	horizontal image resolution in pixels
required	height	int	vertical image resolution in pixels
optional	viewMatrix	16 floats	4x4 view matrix, see computeViewMatrix*
optional	projectionMatrix	16 floats	4x4 projection matrix, see computeProjection*
optional	lightDirection	vec3, list of 3 floats	light direction
optional	lightColor	vec3, list of 3 floats	light color in [RED,GREEN,BLUE] in range 0..1
optional	lightDistance	float	distance of the light
optional	shadow	int	1 for shadows, 0 for no shadows
optional	lightAmbientCoeff	float	light ambient coefficient
optional	lightDiffuseCoeff	float	light diffuse coefficient
optional	lightSpecularCoeff	float	light specular coefficient

getCameraImage returns a list of parameters:

width	int	width image resolution in pixels (horizontal)
height	int	height image resolution in pixels (vertical)
rgbPixels	list of [char RED,char GREEN,char BLUE] [0..width*height]	list of pixel colors in R,G,B format, in range [0..255] for each color
depthPixels	list of float [0..width*height]	depth buffer
segmentationMaskBuffer	list of int [0..width*height]	for each pixels the visible object index

Collision Detection Queries

You can query the contact point information that existed during the last 'stepSimulation'. To get the contact points you can use the 'getContactPoints' API. Note that the 'getContactPoints' will not recompute any contact point information.

getContactPoints

The getContactPoints input parameters are as follows.

optional	filterBodyUniqueIdA	int	only report contact points that involve body A
optional	filterBodyUniqueIdB	int	only report contact points that involve body B

getContactPoints will return a list of contact points. Each contact point has the following fields:

contactFlag	int	reserved
bodyUniqueIdA	int	body unique id of body A
bodyUniqueIdA	int	body unique id of body B
linkIndexA	int	link index of body A, -1 for base
linkIndexB	int	link index of body B, -1 for base

positionOnA	vec3, list of 3 floats	contact position on A, in Cartesian world coordinates
positionOnB	vec3, list of 3 floats	contact position on B, in Cartesian world coordinates
contactNormalOnB	vec3, list of 3 floats	contact normal on B, pointing towards A
contactDistance	float	contact distance, positive for separation, negative for penetration
normalForce	float	normal force applied during the last 'stepSimulation'

getClosestPoints

It is also possible to compute the closest points, independent from stepSimulation. This also lets you compute closest points of objects with an arbitrary separating distance. In this query there will be no normal forces reported.

Inverse Dynamics

todo

Inverse Kinematics

todo

Build and install pybullet

There are a few different ways to install pybullet on Windows, Mac OSX and Linux. First get the source code from github, using

git clone https://github.com/bulletphysics/bullet3

Using cmake on Linux and Mac OSX

0) download and install cmake

1) Run the shell script in the root of Bullet:

build_and_run_cmake_pybullet_double.sh

2) Make sure Python finds our pybullet.so module:
export PYTHONPATH = /your_path_to_bullet/build_cmake/examples/pybullet

That's it. Test pybullet by running a python interpreter and do 'import pybullet'

Using premake for Windows

Make sure some Python version is installed in c:\python-3.5.2 (or other version folder name)

Click on build_visual_studio_vr_pybullet_double.bat and open the 0_Bullet3Solution.sln project in Visual Studio, convert projects if needed.

Switch to Release mode, and compile the 'pybullet' project.
Then there are a few options to import pybullet in a Python interpreter:

export PYTHONPATH=c:\develop\bullet3\bin
(replace with actual folder where Bullet is)

or create an administrator prompt (cmd.exe) and create a symbolic link as follows
cd c:\python-3.5.2\dlls
mklink pybullet.pyd c:\develop\bullet3\bin\pybullet_vs2010.dll

Then run python.exe and import pybullet should work.

TODO: Using setup.py and pip easy installation.