pybullet quickstart guide

Erwin Coumans, 2017
Check most up-to-date Google Docs [version online](#) ,comments are welcome 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 and ray intersection 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
physicsClient = 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)
disconnect()
```

connect, disconnect

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 physics server returning the status. pybullet has some build-in physics servers: DIRECT and GUI.

connect using DIRECT, 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_M EMORY, 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

connect returns a physics client id or -1 if not connected. The physics client Id is an optional argument to most of the other pybullet commands. If you don't provide it, it will assume physics client id = 0. You can connect to multiple different physics servers, except for GUI.

connect 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.

connect 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.

disconnect
You can disconnect to a physics server, using the physics client Id, returned by the connect call (if non-negative). A 'DIRECT' or 'GUI' physics server will shutdown. A remove (out-of-process) physics server will keep on running. See also 'resetSimulation' to remove the all and items.

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
optional	physicsClientId	int	if you connect to multiple physics servers, you can pick which one.

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

Important note: most joints (slider, revolute, continuous) have motors enabled by default. This is similar to a robot joint with a very high-friction harmonic drive. You should set the joint motor control mode and target settings using pybullet.setJointMotorControl2. See the setJointMotorControl2 API for more information.

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	Experimental. 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	force the base of the loaded object to be static
optional	physicsClientId	int	if you connected to multiple servers, you can pick one.

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.

loadBullet, loadSDF, loadMJCF

You can also load objects from other file formats, such as .bullet, .sdf and .mjcf. Those file formats support multiple objects, so the return value is a list of object unique ids. The SDF format is explained in detail at http://sdformat.org . The loadSDF command only extracts some essential parts of the SDF related to the robot models and geometry, and ignores many

elements related to cameras, lights and so on. The loadMJCF format is work-in-progress as well, to load assets from MuJoCo, used in OpenAI Gym.
See also the Important note under loadURDF related to default joint motor settings, and make sure to use setJointMotorControl2.

required	fileName	string	a relative or absolute path to the URDF file on the file system of the physics server.
optional	useMaximalCoordinates	int	Experimental. See loadURDF for more details.
optional	physicsClientId	int	if you connected to multiple servers, you can pick one.

saveWorld

You can create a snapshot of the current world as a pybullet Python file, stored on the server. saveWorld can be useful as a basic editing feature, setting up the robot, joint angles, object positions and environment for example in VR. Later you can just load the pybullet Python file to re-create the world. Note that not all settings are stored in the world file at the moment.
The input arguments are:

required	fileName	string	filename of the pybullet file.
optional	clientServerId	int	if you connected to multiple servers, you can pick one

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]

getMatrixFromQuaterion

getMatrixFromQuaterion is a utility API to create a 3x3 matrix from a quaternion. The input is a quaternion and output a list of 9 floats, representing the matrix.

stepSimulation

stepSimulation will perform all the actions in a single forward dynamics simulation step such as collision detection, constraint solving and integration.

stepSimulation input arguments are optional:

optional	physicsClientId	int	if you connected to multiple servers, you can pick one.

stepSimulation has are no return values.

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

setRealTimeSimulation

By default, the physics server will not step the simulation, unless you explicitly send a 'stepSimulation' command. This way you can maintain control determinism of the simulation. It is possible to run the simulation in real-time by letting the physics server automatically step the simulation according to its real-time-clock (RTC) using the setRealTimeSimulation command. If you enable the real-time simulation, you don't need to call 'stepSimulation'.

The input parameters are:

required	enableRealTimeSimulation	int	0 to disable real-time simulation, 1 to enable
optional	physicsClientId	int	if you connected to multiple servers, you can pick one.

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.
optional	physicsClientId	int	if you connected to multiple servers, you can pick one.

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.

resetBasePositionAndOrientation

You can reset the position and orientation of the base (root) of each object. It is best only to do this at the start, and not during a running simulation, since the command will override the effect of all physics simulation.

The input arguments to resetBasePositionAndOrientation are:

required	objectUniqueId	int	object unique id, as returned from loadURDF.
required	basePosition	vec3	reset the base of the object at the specified position in world space coordinates [X,Y,Z]
required	baseOrientation	vec4	reset the base of the object at the specified orientation as world space quaternion [X,Y,Z,W]
optional	physicsClientId	int	if you connected to multiple servers, you can pick one.

There are no return arguments.

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
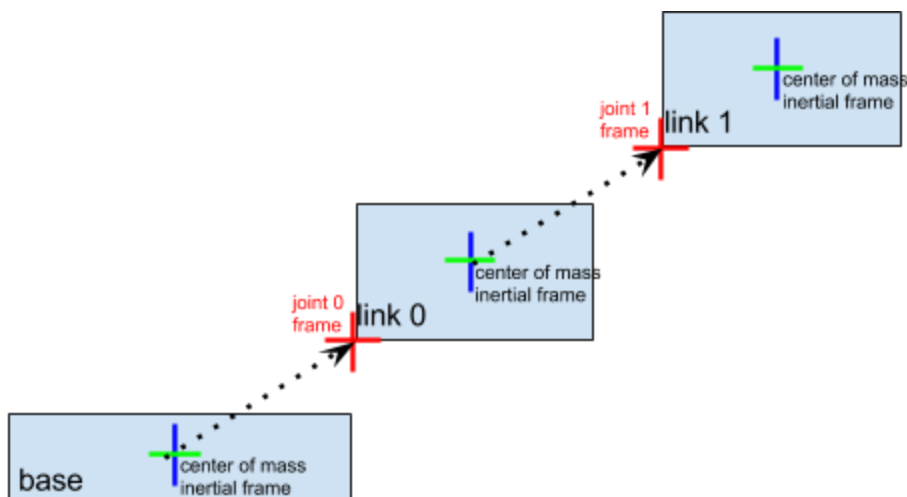
A simulated robot as described in a URDF file has a base, and optionally links connected by joints. Each joint connects one parent link to a child link. At the root of the hierarchy there is a single root parent, that we call base. The base can be either fully fixed, 0 degrees of freedom, or fully free, with 6 degrees of freedom. Since each link is connected to a parent with a single joint, 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. We use the convention that joint frames are expressed relative to the parents center of mass inertial frame, which is aligned with the principle axis of inertia.

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.

optional	physicsClientId	int	if you connected to multiple servers, you can pick one.

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)]
optional	physicsClientId	int	if you connected to multiple servers, you can pick one.

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

setJointMotorControl2

Note: setJointMotorControl is obsolete and replaced by setJointMotorControl2 API.

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.

You can effectively disable the motor by using a force of 0, for example:

maxForce = 0
mode = p.VELOCITY_CONTROL
p.setJointMotorControl(objUid, linkIndex, controlMode=mode, force=maxForce)

If you want a wheel to maintain a constant velocity, with a max force you can use:
maxForce = 500
p.setJointMotorControl(bodyUniqueId=objUid,
linkIndex=0,
controlMode=p.VELOCITY_CONTROL,
targetVelocity = targetVel,
force = maxForce)

The input arguments to setJointMotorControl2 are:

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
optional	targetPosition	float	in POSITION_CONTROL the targetValue is target position of the joint
optional	targetVelocity	float	in VELOCITY_CONTROL and POSITION_CONTROL the targetValue is target velocity of the joint, see implementation note below.
optional	force	float	in POSITION_CONTROL and VELOCITY_CONTROL this is the maximum motor force used to reach the target value. In TORQUE_CONTROL this is the force/torque to be applied each simulation step.
optional	positionGain	float	See implementation note below
optional	velocityGain	float	See implementation note below
optional	physicsClientId	int	if you connected to multiple servers, you can pick one.

Note: the actual implementation of the joint motor controller is as a constraint for POSITION_CONTROL and VELOCITY_CONTROL, and as an external force for TORQUE_CONTROL:

method	implementation	component	constraint error to be minimized

POSITION_CONTROL	constraint	velocity and position constraint	error = position_gain*(desired_position-actual_position)+velocity_gain*(desired_velocity-actual_velocity)
VELOCITY_CONTROL	constraint	pure velocity constraint	error = desired_velocity - actual_velocity
TORQUE_CONTROL	external force		

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)]
optional	physicsClientId	int	if you connected to multiple servers, you can pick one.

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

resetJointState

You can reset the state of the joint. It is best only to do this at the start, while not running the simulation: resetJointState overrides all physics simulation.

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

required	targetValue	float	the joint position (angle)
optional	physicsClientId	int	if you connected to multiple servers, you can pick one.

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	
optional	physicsClientId	int	if you connected to multiple servers, you can pick one.

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.

Example scripts (could be out-of-date, check actual examples/pybullet folder.)

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.
examples/pybullet/inverse_kinematics.py	Show how to use the calculateInverseKinematics command, creating a Kuka ARM clock
Example pybullet for OpenAI GYM, DeepMind Lab	work-in-progress. See cartpole++ http://matpalm.com/blog/cartpole_plus_plus/ https://github.com/matpalm/cartpoleplusplus

getBaseVelocity, resetBaseVelocity

You get access the linear and angular velocity of the base of a body using getBaseVelocity. The input parameters are:

required	bodyUniqueId	int	body unique id, as returned from the load* methods.
optional	physicsClientId	int	if you connected to multiple servers, you can pick one.

This returns a list of two vector3 values (3 floats in a list) representing the linear velocity [x,y,z] and angular velocity [wx,wy,wz] in Cartesian worldspace coordinates.

You can reset the linear and/or angular velocity of the base of a body using resetBaseVelocity. The input parameter are:

required	objectUniqueId	int	body unique id, as returned from the load* methods.
optional	linearVelocity	vec3, list of 3 floats	linear velocity [x,y,z] in Cartesian world coordinates.
optional	angularVelocity	vec3, list of 3 floats	angular velocity [wx,wy,wz] in Cartesian world coordinates.
optional	physicsClientId	int	if you connected to multiple servers, you can pick one.

applyExternalForce, applyExternalTorque

You can apply a force or torque to a body using applyExternalForce and applyExternalForce. Note that this method will only work when explicitly stepping the simulation using

stepSimulation, in other words: setRealTimeSimulation(0). After each simulation step, the external forces are cleared to zero. If you are using 'setRealTimeSimulation(1), applyExternalForce/Torque will have undefined behavior (either 0, 1 or multiple force/torque applications).

required	objectUniqueId	int	object unique id as returned by load methods.
required	linkIndex	int	link index or -1 for the base.
required	forceObj	vec3, list of 3 floats	force vector to be applied [x,y,z]. See flags for coordinate system.
required	posObj	vec3, list of 3 floats	position on the link where the force is applied. See flags for coordinate system.
required	flags	int	Specify the coordinate system of force/position: either WORLD_FRAME for Cartesian world coordinates or or LINK_FRAME for local link coordinates.
optional	physicsClientId	int	

getNumBodies, getBodyInfo, getBodyUniqueId

getNumBodies will return the total number of bodies in the physics server.
If you used 'getNumBodies' you can query the body unique id's using 'getBodyUniqueId'. Note that all APIs already return body unique id's, so you typically never need to use getBodyUniqueId if you keep track of them.

getBodyInfo will return the base name, as extracted from the URDF, SDF, MJCF or other file.

createConstraint, removeConstraint, changeConstraint

URDF, SDF and MJCF specify articulated bodies as a tree-structures without loops. The 'createConstraint' allows you to connect specific links of bodies to close those loops. See Bullet/examples/pybullet/quadruped.py how to connect the legs of a quadruped 5-bar closed loop linkage. In addition, you can create arbitrary constraints between objects, and between an object and a specific world frame. See Bullet/examples/pybullet/constraint.py for an example.
It can also be used to control the motion of physics objects, driven by animated frames, such as a VR controller. It is better to use constraints, instead of setting the position or velocity directly for such purpose, since those constraints are solved together with other dynamics constraints.

createConstraint has the following input parameters:

required	parentBodyUniqueId	int	parent body unique id
required	parentLinkIndex	int	parent link index (or -1 for the base)
required	childBodyUniqueId	int	child body unique id, or -1 for no body (specify a non-dynamic child frame in world coordinates)
required	childLinkIndex	int	child link index, or -1 for the base
required	jointType	int	joint type: JOINT_REVOLUTE, JOINT_PRISMATIC, JOINT_FIXED, JOINT_POINT2POINT
required	jointAxis	vec3, list of 3 floats	joint axis, in child link frame
required	parentFramePosition	vec3, list of 3 floats	position of the joint frame relative to parent center of mass frame.
required	childFramePosition	vec3, list of 3 floats	position of the joint frame relative to a given child center of mass frame (or world origin if no child specified)
optional	parentFrameOrientation	vec4, list of 4 floats	the orientation of the joint frame relative to parent center of mass coordinate frame
optional	childFrameOrientation	vec4, list of 4 floats	the orientation of the joint frame relative to the child center of mass coordinate frame (or world origin frame if no child specified)
optional	physicsClientId	int	if you connected to multiple servers, you can pick one.

createConstraint will return an integer unique id, that can be used to change or remove the constraint.

changeConstraint allows you to change parameters of an existing constraint. The input parameters are:

required	userConstraintUniqueId	int	unique id returned by createConstraint
optional	jointChildPivot	vec3, list of 3 floats	updated child pivot, see 'createConstraint'
optional	jointChildFrameOrientation	vec4, list of 4 floats	updated child frame orientation as quaternion
optional	maxForce	float	maximum force that constraint can apply
optional	physicsClientId	int	if you connected to multiple servers, you can pick one.

See also Bullet/examples/pybullet/constraint.py

removeConstraint will remove the a constraint, given its unique id. Its input parameters are:

required	userConstraintUniqueId	int	unique id as returned by createConstraint
optional	physicsClientId	int	unique id as returned by 'connect'

getNumConstraints

You can query for the total number of constraints, created using 'createConstraint'. Optional parameter is the int physicsClientId.

getConstraintInfo

You can query the constraint info give a constraint unique id.
The input parameters are

required	constraintUniqueId	int	unique id as returned by createConstraint
optional	physicsClientId	int	unique id as returned by 'connect'

The output list is:

parentBodyIndex	int	See createConstraint
parentJointIndex	int	See createConstraint
childBodyIndex	int	See createConstraint
childLinkIndex	int	See createConstraint
constraintType	int	See createConstraint
jointAxis	vec3, list of 3 floats	See createConstraint
jointPivotInParent	vec3, list of 3 floats	See createConstraint
jointPivotInChild	vec3, list of 3 floats	See createConstraint
jointFrameOrientationParemt	vec4, list of 4 floats	See createConstraint
jointFrameOrientationChild	vec4, list of 4 floats	See createConstraint
maxAppliedForce	float	See createConstraint

setTimeStep

You can set the physics engine timestep that is used when calling 'stepSimulation'. It is best to only call this method at the start of a simulation. Don't change this time step regularly. setTimeStep can also be achieved using the new setPhysicsEngineParameter API.

The input parameters are:

required	timeStep	float	Each time you call 'stepSimulation' the timeStep will proceed with 'timeStep'.
optional	physicsClientId	int	if you connected to multiple servers, you can pick one.

setPhysicsEngineParameter

You can set physics engine parameters using the setPhysicsEngineParameter API. The following input parameters are exposed:

optional	fixedTimeStep	float	physics engine timestep in fraction of seconds, each time you call 'stepSimulation'. Same as 'setTimeStep'
optional	numSolverIterations	int	Choose the number of constraint solver iterations.
optional	useSplitImpulse	int	Advanced feature, only when using maximial coordinates: split the positional constraint solving and velocity constraint solving in two stages, to prevent huge penetration recovery forces.
optional	splitImpulsePenetrationThreshold	float	Related to 'useSplitImpulse': if the penetration for a particular contact constraint is less than this specified threshold, no split impulse will happen for that contact.
optional	numSubSteps	int	Subdivide the physics simulation step further by 'numSubSteps'. This will trade performance over accuracy.
optional	collisionFilterMode	int	Use 0 for default collision filter: (group A&maskB) AND (groupB&maskA). Use 1 to switch to the OR collision filter: (group A&maskB) OR (groupB&maskA)
optional	physicsClientId	int	if you connected to multiple servers, you can pick one.

resetSimulation

resetSimulation will remove all objects from the world and reset the world to initial conditions. It takes one optional parameter: the physics client Id (in case you created multiple physics server connections).

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	left screen (canvas) coordinate
required	right	float	right screen (canvas) coordinate
required	bottom	float	bottom screen (canvas) coordinate
required	top	float	top screen (canvas) coordinate
required	near	float	near plane distance
required	far	float	far plane distance

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

computeProjectionMatrixFOV

This command also will return a 4x4 projection matrix, using different parameters. You can check out OpenGL documentation for the meaning of the parameters.
The input parameters are:

required	fov	float	field of view
required	aspect	float	aspect ratio
required	nearVal	float	near plane distance
required	farVal	float	far plane distance

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. Note that pybullet can be compiled using the numpy option: using numpy will improve the performance of copying the camera pixels from C to Python. Note: the old renderImage API is obsolete and replaced by getCameraImage.

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
optional	physicsClientId	int	if you connected to multiple servers, you can pick one.

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

getVisualShapeData

You can access visual shape information using getVisualShapeData. You could use this to bridge your own rendering method with pybullet simulation, and synchronize the world transforms manually after each simulation step.
The input parameters are:

required	objectUniqueId	int	object unique id, as returned by a load method.
optional	physicsClientId	int	physics client id as returned by 'connect'

The output is a list of visual shape data, each visual shape is in the following format:

objectUniqueId	int	object unique id, same as the input
linkIndex	int	link index or -1 for the base
visualGeometryType	int	visual geometry type (TBD)
dimensions	vec3, list of 3 floats	dimensions (size, local scale) of the geometry
meshAssetFileName	string, list of chars	path to the triangle mesh, if any. Typically relative to the URDF, SDF or MJCF file location, but could be absolute.
localInertiaFrame position	vec3, list of 3 floats	TBD see note
localInertiaFrame orientation	vec4, list of 4 floats	TBD, see note

Note: the visual frame (missing) is relative to the links joint frame, not center of mass. So you need to multiply with inverse local inertial frame to use the center of mass transform returned by 'getJointState'.

resetVisualShapeData

You can use resetVIsualShapeData to change the texture of a shape. This texture will currently only affect the software renderer (see getCameraImage), not the OpenGL visualization window (yet).

required	objectUniqueId	int	object unique id, as returned by load method.
required	jointIndex	int	link index
required	shapeIndex	int	shape index, within range. See getVisualShapeData.
required	textureUniqueId	int	texture unique id, as returned by 'loadTexture' method

required	physicsClientId	int	physics client id as returned by 'connect'

loadTexture

Load a texture from file and return a non-negative texture unique id if the loading succeeds. This unique id can be used with 'resetVisualShapeData.

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.

getOverlappingObjects

This query will return all the unique ids of objects that have axis aligned bounding box overlap with a given axis aligned bounding box.

required	aabbMin	vec3, list of 3 floats	minimum coordinates of the aabb
required	aabbMax	vec3, list of 3 floats	minimum coordinates of the aabb
optional	physicsClientId	int	if you connected to multiple servers, you can pick one.

The getOverlappingObjects will return a list of object unique ids.

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
optional	physicsClientId	int	if you connected to multiple servers, you can pick one.

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.

getClosestPoints input parameters:

required	objectUniqueIdA	int	object unique id for first object (A)
required	objectUniqueIdB	int	object unique id for second object (B)
required	maxDistance	float	If the distance between objects exceeds this maximum distance, no points may be returned.
optional	physicsClientId	int	if you connected to multiple servers, you can pick one.

getClosestPoints returns a list of closest points in the same format as getContactPoints (but normalForce is always zero in this case)

rayTest

You can perform a raycast to find the intersection information of the first object hit.

required	rayFromPosition	vec3, list of 3 floats	start of the ray in world coordinates
required	rayToPosition	vec3, list of 3 floats	end of the ray in world coordinates
optional	physicsClientId	int	if you connected to multiple servers, you can pick one.

The raytest query will return the following information in case of an intersection:

objectUniqueId	int	object unique id of the hit object
linkIndex	int	link index of the hit object, or -1 if none/parent.
hit fraction	float	hit fraction along the ray in range [0,1] along the ray.
hit position	vec3, list of 3 floats	hit position in Cartesian world coordinates
hit normal	vec3, list of 3 floats	hit normal in Cartesian world coordinates

Inverse Dynamics

calculateInverseDynamics

calculateInverseDynamics will compute the forces needed to reach the given joint accelerations, starting from specified joint positions and velocities.

required	bodyUniqueId	int	body unique id, as returned by loadURDF etc.
required	jointPositions	list of float	joint positions (angles)
required	jointVelocities	list of float	joint velocities
required	jointAccelerations	list of float	desired joint accelerations
optional	physicsClientId	int	if you connected to multiple servers, you can pick one.

calculateInverseDynamics returns a list of joint forces.

Inverse Kinematics

You can compute the joint angles that makes the end-effector reach a given target position in Cartesian world space. Optionally you can also specify the target orientation of the end effector. In addition, there is an option to use the null-space to specify joint limits and rest poses. This optional null-space support requires all 4 lists (lowerLimits, upperLimits, jointRanges, restPoses), otherwise regular IK will be used. See also inverse_kinematics.py example in Bullet/examples/pybullet folder for details.

calculateInverseKinematics

calculateInverseKinematics input parameters are

required	bodyUniqueId	int	body unique id, as returned by loadURDF
required	endEffectorLinkIndex	int	end effector link index
required	targetPosition	vec3, list of 3 floats	target position in Cartesian world space
optional	targetOrientation	vec3, list of 4 floats	target orientation in Cartesian world space, quaternion [x,y,w,z]. If not specified, pure position IK will be used.
optional	lowerLimits	list of floats [0..nDof]	Optional null-space IK requires all 4 lists (lowerLimits, upperLimits, jointRanges, restPoses). Otherwise regular IK will be used.
optional	upperLimits	list of floats [0..nDof]	
optional	jointRanges	list of floats [0..nDof]	
optional	restPoses	list of floats [0..nDof]	
optional	physicsClientId	int	if you connected to multiple servers, you can pick one.

calculateInverseKinematics returns a list of joint positions. See Bullet/examples/pybullet/inverse_kinematics.py for an example.

Virtual Reality

When pybullet is connected to a virtual reality physics server, you can get access to the VR controller state. The VR physics server uses the OpenVR API for HTC Vive and Oculus Rift Touch controller support. OpenVR is currently working on Windows, Valve is also working on a [Linux version](.).

See also for an example video of the VR example, part of Bullet, that can be fully controlled using pybullet over shared memory or UDP connection.

getVREvents

getVREvents will return a list of controllers that changed state since the last call to getVREvents. getVREvents has one optional input parameter, physics client Id. The output parameters are:

controllerId	int	controller index (0..MAX_VR_CONTROLLERS)
controllerPosition	vec3, list of 3 floats	controller position, in world space Cartesian coordinates
controllerOrientation	vec4, list of 4 floats	controller orientation quaterion [x,y,z,w] in world space
controllerAnalogueAxis	float	analogue axis value
numButtonEvents	int	number of button events since last call to getVREvents
numMoveEvents	int	number of move events since last call to getVREvents
buttons	int[8], list of 8 integers	flags for each button: VR_BUTTON_IS_DOWN (currently held down), VR_BUTTON_WAS_TRIGGERED (went down at least once since last cal to getVREvents, VR_BUTTON_WAS_RELEASED (was released at least once since last call to getVREvents). Note that only VR_BUTTON_IS_DOWN reports actual current state. For example if the button went down and up, you can tell from the RELEASE/TRIGGERED flags, even though IS_DOWN is still false.

See Bullet/examples/pybullet/vrEvents.py for an example of VR drawing.

setVRCameraState

setVRCameraState allows to set the camera root transform offset position and orientation. This allows to control the position of the VR camera in the virtual world. It is also possible to let the VR Camera track an object, such as a vehicle.

setVRCameraState has the following arguments (there are no return values).

optional	rootPosition	vec3, vector of 3 floats	camera root position
optional	rootOrientation	vec3, vector of 3 floats	camera root orientation

optional	trackObject	vec3, vector of 3 floats	the object unique id to track
optional	physicsClientId	int	if you connected to multiple servers, you can pick one.

Debug GUI, Lines, Text, Parameters

pybullet has some functionality to make it easier to debug, visualize and tune the simulation. This feature is only useful if there is some 3d visualization window, such as GUI mode or when connected to a separate physics server (such as Example Browser in 'Physics Server' mode or standalone Physics Server with OpenGL GUI).

addUserDebugLine

You can add a 3d line specified by a 3d starting point (from) and end point (to), a color [red,green,blue], a line width and a duration in seconds. The arguments to addUserDebugline are:

required	lineFromXYZ	vec3, list of 3 floats	starting point of the line in Cartesian world coordinates
required	lineToXYZ	vec3, list of 3 floats	end point of the line in Cartesian world coordinates
optional	lineColorRGB	vec3, list of 3 floats	RGB color [Red, Green, Blue] each component in range [0..1]
optional	lineWidth	float	line width (limited by OpenGL implementation)
optional	lifeTime	float	use 0 for permanent line, or positive time in seconds (afterwards the line with be removed automatically)
optional	physicsClientId	int	if you connected to multiple servers, you can pick one

addUserDebugLine will return a non-negative unique id, that lets you remove the line using removeUserDebugItem.

addUserDebugText

You can add some 3d text at a specific location using a color and size. The input arguments are:
"text", "textPosition", "textColorRGB", "textSize", "lifeTime","physicsClientId"

required	text	text	text represented as a string (array of characters)

required	textPosition	vec3, list of 3 floats	3d position of the text in Cartesian world coordinates [x,y,z]
optional	textColorRGB	vec3, list of 3 floats	RGB color [Red, Green, Blue] each component in range [0..1]
optional	textSize	float	line width (limited by OpenGL implementation)
optional	lifeTime	float	use 0 for permanent line, or positive time in seconds (afterwards the line with be removed automatically)
optional	physicsClientId	int	if you connected to multiple servers, you can pick one

addUserDebugText will return a non-negative unique id, that lets you remove the line using removeUserDebugItem.

addUserDebugParameter, readUserDebugParameter

addUserDebugParameter lets you add custom sliders to tune parameters. It will return a unique id. This lets you read the value of the parameter using readUserDebugParameter. The input parameters of addUserDebugParameter are:

required	paramName	string	name of the parameter
required	rangeMin	float	minimum value
required	rangeMax	float	maximum value
required	startValue	float	starting value
optional	physicsClientId	int	if you connected to multiple servers, you can pick one

The input parameters of readUserDebugParameter are:

required	itemUniqueId	int	the unique id returned by 'addUserDebugParameter)
optional	physicsClientId	int	if you connected to multiple servers, you can pick one

Return value is the most up-to-date reading of the parameter.

removeUserDebugItem

The functions to add user debug lines, text or parameters will return a non-negative unique id if it succeeded. You can remove the debug item using this unique id using the removeUserDebugItem method.The input parameters are:

required	itemUniqueId	int	unique id of the debug item to be removed (line, text etc)
optional	physicsClientId	int	if you connected to multiple servers, you can pick one

setDebugObjectColor

The built-in OpenGL visualizers have a wireframe debug rendering feature: press 'w' to toggle. The wireframe has some default colors. You can override the color of a specific object and link using setDebugObjectColor. The input parameters are:

required	objectUniqueId	int	unique id of the object
required	linkIndex	int	link index
optional	objectDebugColorRGB	vec3, list of 3 floats	debug color in [Red,Green,Blue]. If not provided, the custom color will be removed.
optional	physicsClientId	int	if you connected to multiple servers, you can pick one

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) On Mac OSX, create symbolic link:
   ln -s pybullet.dylib pybullet.so
3) 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 enter 'import pybullet' to see if the module loads. If so, you can play with the pybullet scripts in Bullet/examples/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:

1) Rename pybullet_vs2010..dll to pybullet.pyd and start the Python.exe interpreter using bullet/bin as the current working directory. Optionally for debugging: rename bullet/bin/pybullet_vs2010_debug.dll to pybullet_d.pyd and start python_d.exe)
2) Rename bullet/bin/pybullet_vs2010..dll to pybullet.pyd and use command prompt:
   export PYTHONPATH=c:\develop\bullet3\bin (replace with actual folder where Bullet is) or create this PYTHONPATH environment variable using Windows GUI
3) 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 easier installation of pybullet.