CS4910: Deep Learning for Robotics

David Klee
klee.d@northeastern.edu

T/F, 3:25-5:05pm
Behrakis Room 204

https://www.ccs.neu.edu/home/dmklee/cs4910_s22/index.html

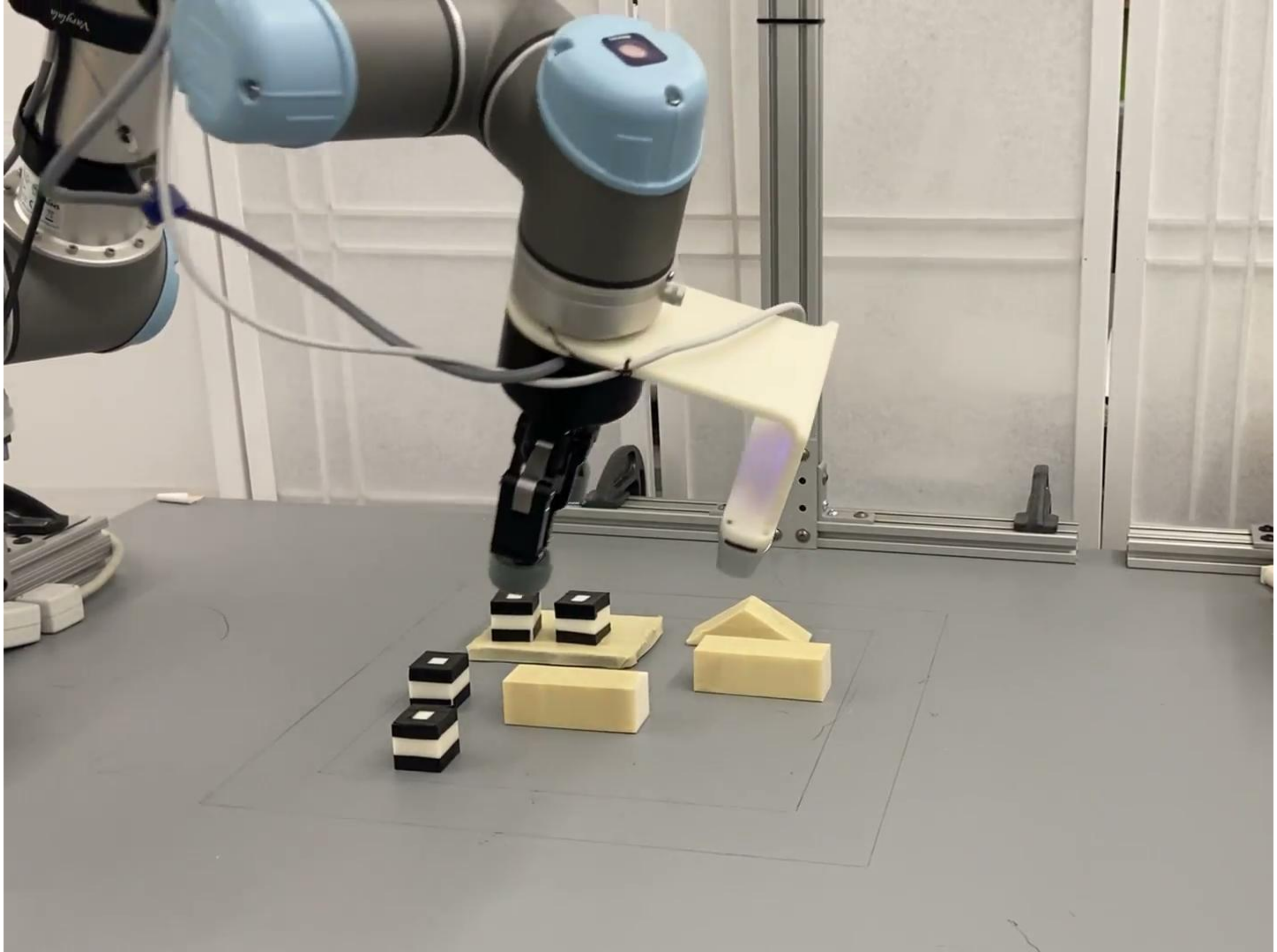
https://piazza.com/northeastern/spring2022/cs4910a/home

Simulating Robots

Today's Agenda

1. Understand how robots are modeled in simulators
2. Learn commands for interacting with robot
3. Compete in a drag race
4. Learn how to place sensors in simulator
5. Hand out robots

Why use a simulator?



Why use a simulator?

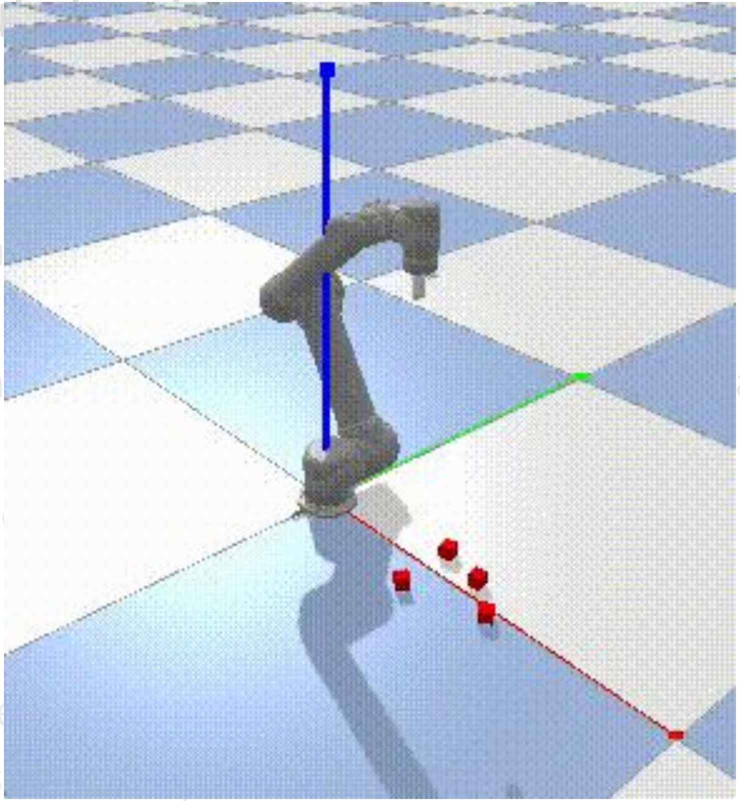
The Good:

- Much cheaper than real world
- Some algorithms are not safe to run on real robot
- Faster ideation and testing of new setups or robot parts
- Data collection is ***much*** faster
- Access to privileged information

The Bad:

- Difficult to achieve photorealism
- Contact dynamics are not perfect
- Researchers are less impressed

The Ugly:



Why use a simu

The Good:

- Much cheaper tha
- Some algorithms
- Faster ideation an arts
- Data collection is
- Access to privileg

The Bad:

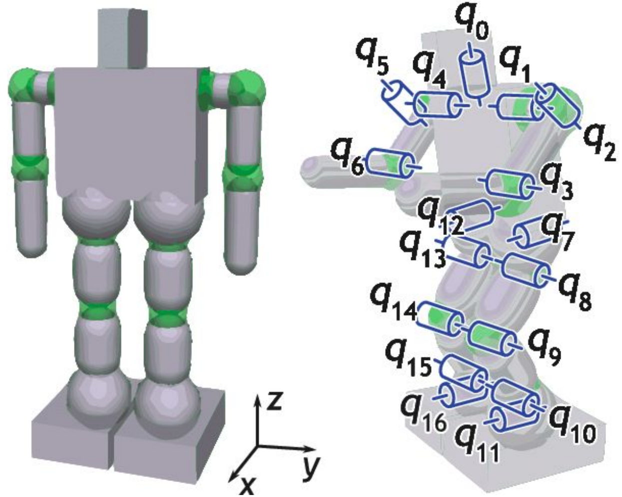
- Difficult to achieve
- Contact dynamics
- Researchers are less impressed

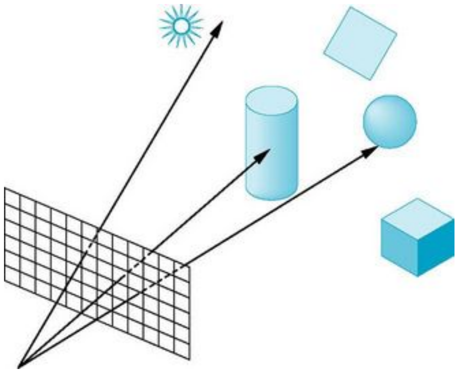
Simulators in a nutshell

1. Simulating Physics (*rigid-body*)
2. Rendering

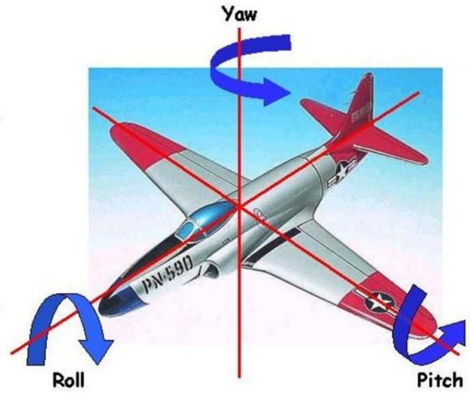
Abstractions to create Environments/Tasks

May support Motion Planning





Rotations and Translations



Talk about Euler angles, quaternions, rotation matrices

Roto-translation matrices

Transformation Matrices

$$\mathbf{T} \in \mathcal{R}^{4\times4} \leftarrow \text{transformation matrix}$$

$$\mathbf{R} \in \mathcal{R}^{3\times3} \leftarrow \text{rotation matrix}$$

$$\vec{t} \in \mathcal{R}^{3} \leftarrow \text{translation vector}$$

$$\tilde{x} = \begin{bmatrix} \vec{x} \\ 1 \end{bmatrix} \in \mathcal{R}^{4} \leftarrow \text{homogeneous vector}$$

$$\mathbf{T} = \left[\begin{array}{c|c} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{array}\right]$$

$$\tilde{x}' = \mathbf{T}\tilde{x} = \mathbf{R}\vec{x} + \vec{t}$$

in-class exercise:

$$T^{-1} = ?$$

Transformation Matrices

$$\mathbf{T} \in \mathcal{R}^{4\times4} \leftarrow \text{transformation matrix}$$

$$\mathbf{R} \in \mathcal{R}^{3\times3} \leftarrow \text{rotation matrix}$$

$$\vec{t} \in \mathcal{R}^{3} \leftarrow \text{translation vector}$$

$$\tilde{x} = \begin{bmatrix} \vec{x} \\ 1 \end{bmatrix} \in \mathcal{R}^{4} \leftarrow \text{homogeneous vector}$$

$$\mathbf{T} = \left[\begin{array}{c|c} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{array} \right]$$

$$\tilde{x}' = \mathbf{T}\tilde{x} = \mathbf{R}\vec{x} + \vec{t}$$

$$\mathbf{T}^{-1} = \left[\begin{array}{c|c} \mathbf{R}^T & -\mathbf{R}^T\mathbf{t} \\ \mathbf{0}^T & 1 \end{array} \right]$$

Make example of rotation in xy plane

T =

\left[{\begin{array}{c|c}

Furthering your understanding

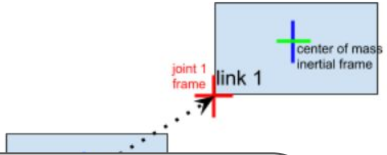
[Scipy.spatial.rotations Library]

Visualizing transformations:

$ python examples/rotation_translation_visualizer.py

A robot is described as a set of links and joints

Links : physical body with appearance, inertia and collision shape



Joints : constraint on how two links interact

(fixe

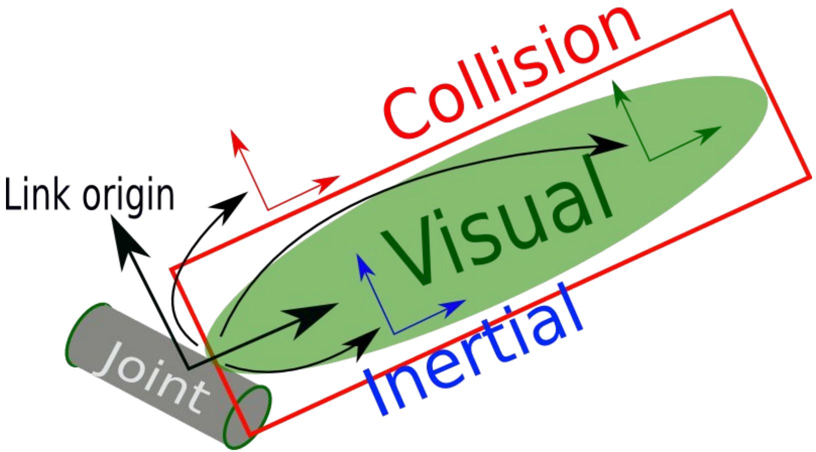
Information on links and joints are described in a Unified Robot Description Format (URDF), and saved as a ".urdf" file.

Onl

- the links must form a 'tree'
- # links = # joints

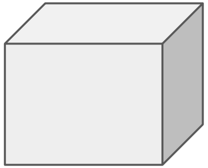
Links

```
<link name="my_link">
  <inertial>
    <origin xyz="0 0 0.5" rpy="0 0 0"/>
    <mass value="1"/>
    <inertia ixx="100"  ixy="0"  ixz="0" iyy="100" iyz="0" izz="100" />
  </inertial>

  <visual>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <box size="1 1 1" />
    </geometry>
    <material name="Cyan">
      <color rgba="0 1.0 1.0 1.0"/>
    </material>
  </visual>

  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <cylinder radius="1" length="0.5"/>
    </geometry>
  </collision>
</link>
```

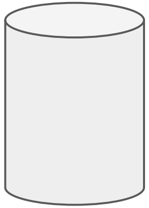


Slide material from: http://wiki.ros.org/urdf/XML/link

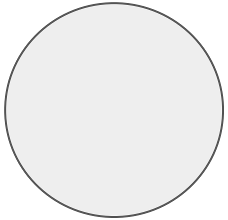
Types of geometry



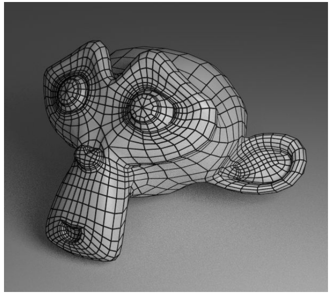
box
size reflects side length
origin at center



cylinder
has radius and length
origin at center

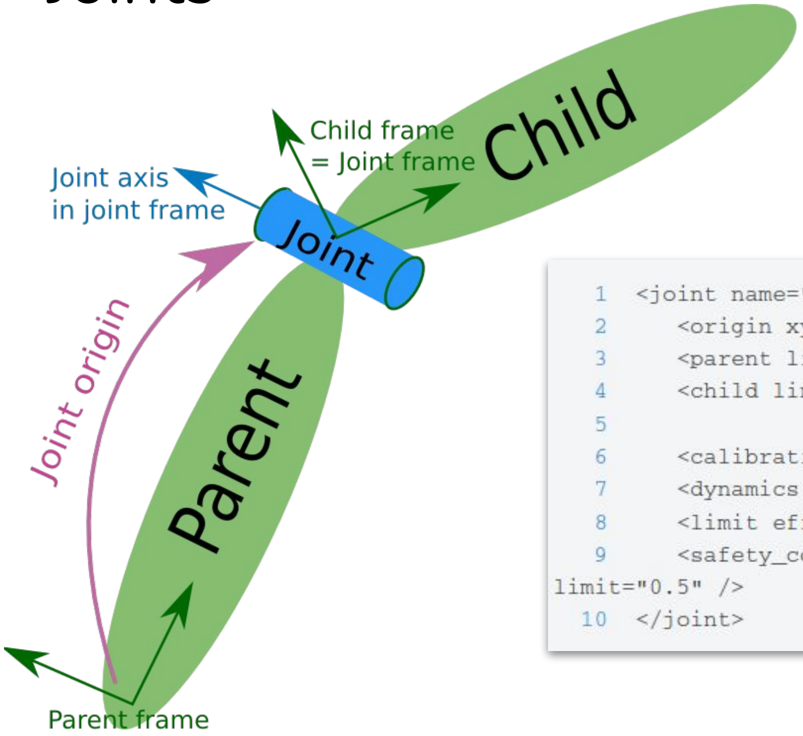


sphere
has radius
origin at center



mesh
uses .obj or .dae file
can be slow for collision checking

Joints




```
<joint name="my_joint" type="floating">
   <origin xyz="0 0 1" rpy="0 0 3.1416"/>
   <parent link="link1"/>
   <child link="link2"/>

   <calibration rising="0.0"/>
   <dynamics damping="0.0" friction="0.0"/>
   <limit effort="30" velocity="1.0" lower="-2.2" upper="0.7" />
   <safety_controller k_velocity="10" k_position="15" soft_lower_limit="-2.0" soft_upper_limit="0.5" />
</joint>
```

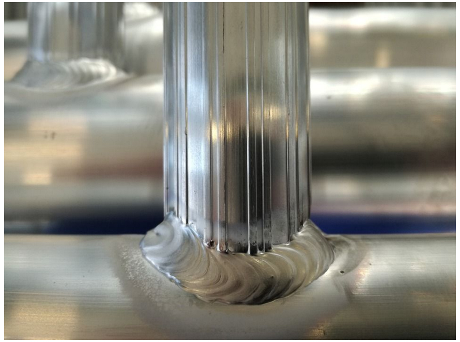
Slide material from: http://wiki.ros.org/urdf/XML/joint

Types of Joints

Most common for robotic arms

revolute
has upper, lower limits

continuous
no limits



fixed
no movement

prismatic
slides along single axis, has upper, lower limits

floating
rarely used, since there are no constraints

Relevant Pybullet Commands

pb.getNumJoints -> gets number of joints/links

pb.getJointInfo -> allows you to match joint/link names to ids

pb.resetBasePositionAndOrientation -> moves base of robot

pb.getLinkStates -> provides pose and velocity of robot links

pb.getJointState -> provides joint position, velocity, and forces

For details on args and output, see Quickstart Guide

Drag racing in Pybullet

Play around with URDF, points for speed and style:

$ python examples/drag_racing.py

You can change anything but MAX_FORCE

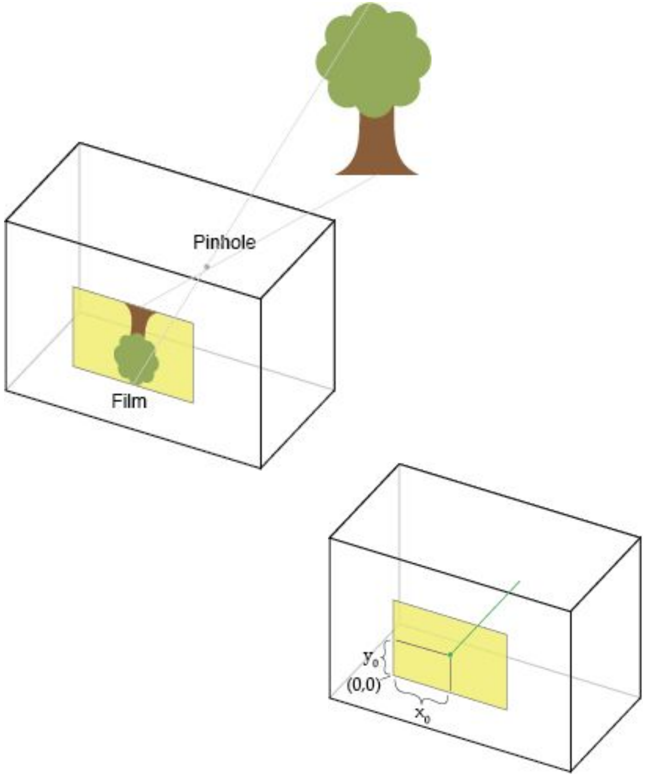
For more details...

General Tutorials [using ROS]

Parametrizing URDF with xacro

Creating Robots programmatically with Pybullet\

Rendering images with virtual camera



Intrinsic Matrix (Projection Matrix)

$$K = \begin{pmatrix} f_x & s & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$f_x = f_y = \text{focal length}$$

$$x_0, y_0 = \text{center of pixel space}$$

Material taken from: http://ksimek.github.io/2013/08/13/intrinsic/

Rendering images with virtual camera

Extrinsic Matrix (View Matrix)

$$V = \left[\mathbf{R} \,\middle|\, \vec{t} \right]$$

$\mathbf{R}$ = rotation from world to camera

$\vec{t}$ = translation from world to camera

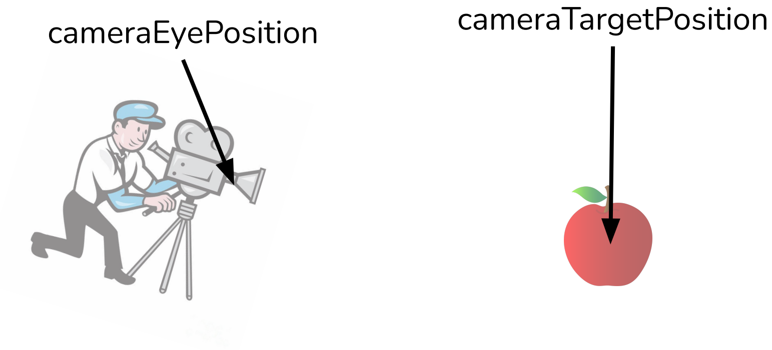
Material taken from: http://ksimek.github.io/2013/08/13/intrinsic/

Converting world point to pixel space

$$\vec{x}_{pixel} = \mathbf{KV}\vec{x}_{world}$$

Material taken from: http://ksimek.github.io/2013/08/13/intrinsic/

View Matrix in Pybullet (Two ways)



(cameraUpVector is usually <0,1,0>)

```
pb.computeViewMatrix(cameraEyePosition,
                     cameraTargetPosition,
                     cameraUpVector)
```

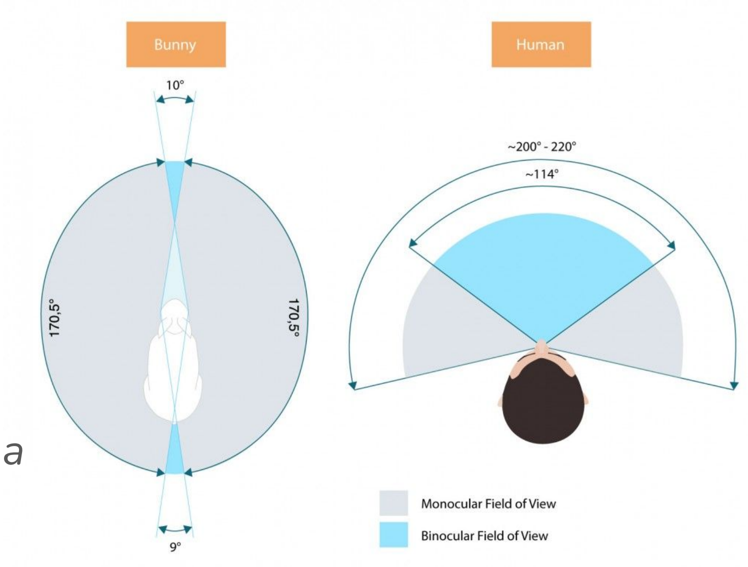
```
pb.computeViewMatrixFromYawPitchRoll(
    cameraTargetPosition,
    distance,
    yaw,
    pitch,
    roll,
    upAxisIndex=2,
)
```

View Matrix is a flattened version of the full transformation matrix that converts from world to camera coordinates:

T_world2cam = np.reshape(view_mtx, (4,4), order='F')

Projection Matrix in Pybullet

```
pb.computeProjectionMatrixFOV(fov, #degrees
                              aspect=1,
                              nearVal,
                              farVal)
```



nearVal & farVal are especially important if you are using a depth sensor since they define the range of output

Rendering an Image in Pybullet

```
w, h, rgb_img, depth_img, seg_img = pb.getCameraImage(width,
                                                      height,
                                                      view_matrix,
                                                      proj_matrix,
                                                      **kwargs)
```

More information about rendering

Useful discussion of how pybullet constructs intrinsic matrix

Script for placing camera in Pybullet with GUI:

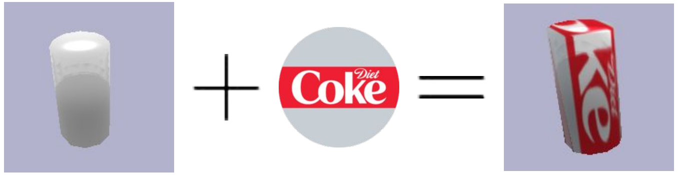
```
$ python examples/easy_pybullet_camera_placement.py
```

Changing Visual Appearance

To avoid creating new URDF’s for all possible appearances, it is often easier to use Python commands

Note: rgbaColor will not work if a texture is already loaded

Note: for advanced textures, it is best to use 3D design software

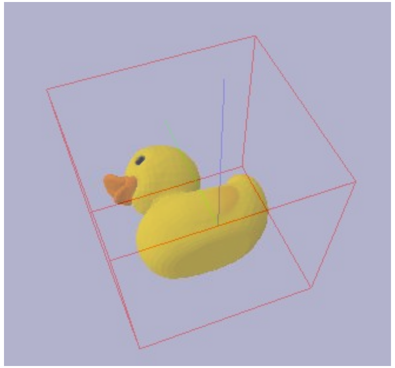



```
pb.loadTexture(“example_texture.png”)
pb.changeVisualShape(objectUniqueId: int, linkIndex: int,
                     textureUniqueId: int,
                     rgbaColor: vec4)
```

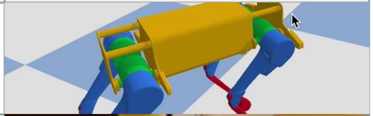
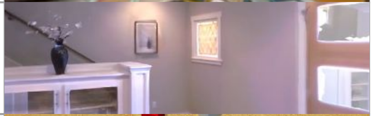
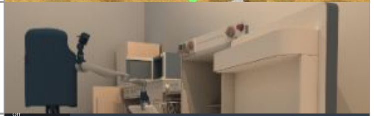
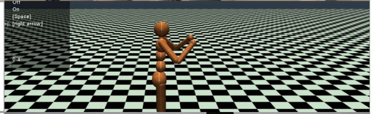
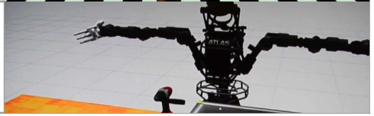
Detecting Collisions

- Collision information is generated during each `pb.stepSimulation()`
- Alternatively, use `pb.performCollisionDetection()` to avoid stepping simulator
  - useful for motion planning to check *if* a collision might occur
- To access collision information, use `pb.getContactPoints` or `pb.getClosestPoints`
- URDF_USE_SELF_COLLISION in pb.loadURDF

minXYZ, maxXYZ ← pb.getAABB



Selection of alternative simulators for robotics/learning

Name	Maintained by	Uses/Known for	Appearance
PyBullet	Google Brain	This class	
DRAKE	MIT & TRI	Robotics/Optimization	
Habitat	FAIR +	Navigation	
RLBench	Imperial College	Robotic Manipulation	
SAPIEN	UCSD +	Mobile robotics	
Mujoco	DeepMind	RL environments	
Gazebo	OSRF	With ROS	

Survey to provide feedback

https://forms.gle/9pTWc3EAY8LtPvF98

Handing out robots

URDFs : robots first (joints/links, material), then objects

Positioning and rotations

Performing motor commands (with inverse kinematics)

Adjusting dynamics (drag race a car)

Adjusting apperance, adding materials

Sensors : fundamentals of rendering, view matrix, proj matrix

Example (mounting sensor on robot arm)

Example: implement suction cup

Additionals: mention support for soft body physics, custom constraint for walking

Mention other simulators out there