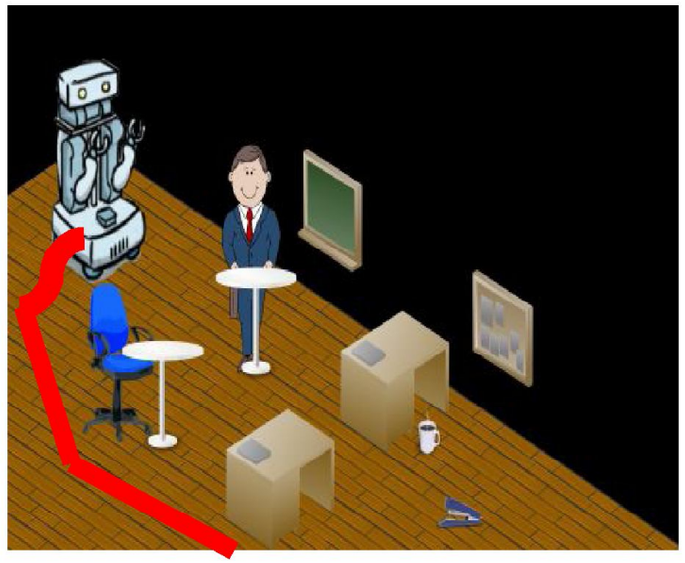
# Contents

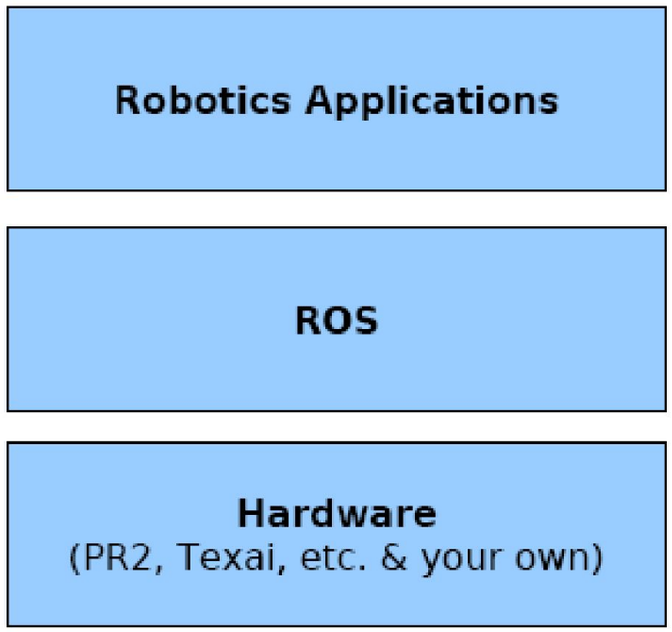
- ROS
  - Basic Structure, Commands etc
- Simulator
- Kinect
- TurtleBot
- Programming Concepts
- OpenCV and PCL
- Conclusion



# ROS

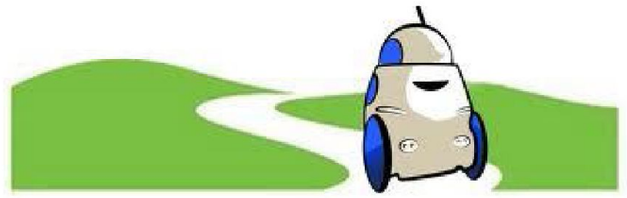
ROS is an **open-source, a software framework for robot software development**, providing operating system-like functionality on a heterogenous computer cluster

- Provides OS services: hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management.
- Based on a graph architecture where processing takes place in nodes
- ROS has two basic "sides"
  - operating system side
  - a suite of user contributed packages




*(Ubuntu recommended)* Linux  Mac OS X  Windows  FreeBSD

# Why ROS



- Code reuse in robotics research and development
- Ready-to-use development environment, comprehensive of manifold tools and client API libraries (C++, Python, Lisp, Java,...)
- Language independent
- Scalable (distributed network of processes loosely coupled)
- Big community and continuous support

# Concept: Don't Reinvent the Wheel

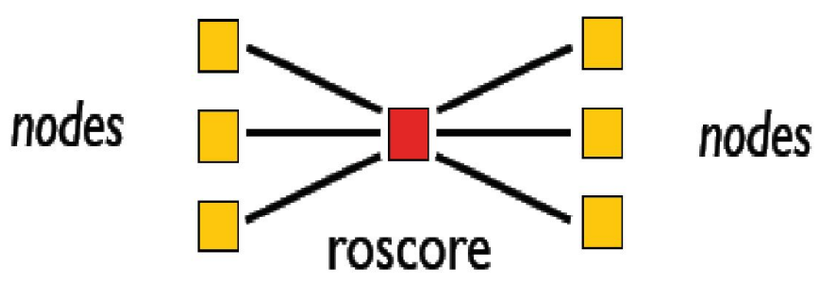
- OpenCV - Computer Vision
- Eigen - Matrix Algebra
- Gazebo - Robot Simulator
- KDL - Kinematics and Dynamics
- TREX - High Level Planning
- PCL – Point Cloud Library
- Many more .....

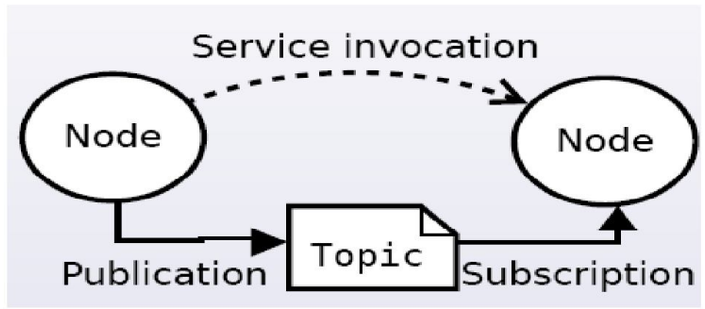
# Goals & Highlights

- ROS is a **distributed framework of processes (aka** Nodes)
- Enables executables to be individually designed and **loosely coupled at runtime**
- Processes can be grouped into Packages and Stacks
- Scaling: ROS is appropriate for large runtime systems and for large development processes
- All ROS core code is **licensed BSD, so it easy to** integrate in your project.
- Many device drivers **and algorithms are available.**
- (Willow Garage also maintains **OpenCV)**

# Concepts: Computation Graph Level

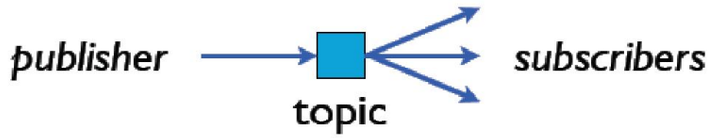
- **Computation Graph: Peer-to-peer network of ROS processes** that are processing data together. The basic Computation Graph concepts of ROS are:
  - nodes,
  - master,
  - parameter server,
  - Services,
  - Messages & topics



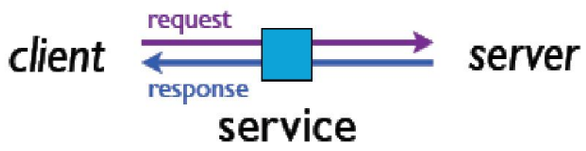



# Key Concepts

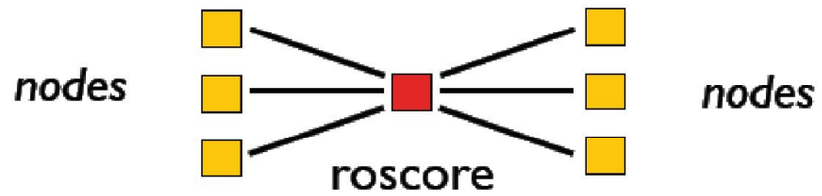
- roscore: Name and Parameter server; singleton
- Node: An agent communicating with ROS
- Nodes communicate via
  - Topics (publish / subscribe) using typed messages




  - Services: Request / Response paradigm method or operation) via typed messages

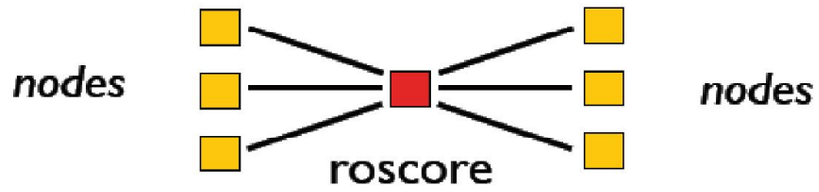


# Nodes



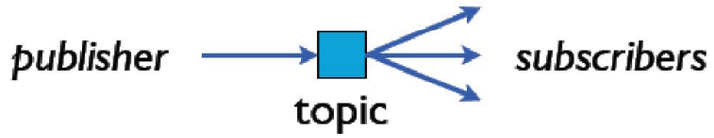

- **Nodes are processes that perform computation.**
- ROS is designed to be modular at a fine-grained scale: a robot control system will usually comprise many nodes.
- For example, one node controls a camera processing, another node performs object recognition.
- A ROS node is written with the **use of a ROS client library,** such as roscpp or rospy.

# Master



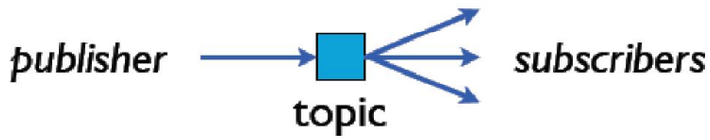

- Provides **name registration and lookup to the rest of the** Computation Graph.
- Without the Master, nodes would not be able to find each other, exchange messages, or invoke services.
- The master stores topics and services, registration information for ROS nodes

# Messages



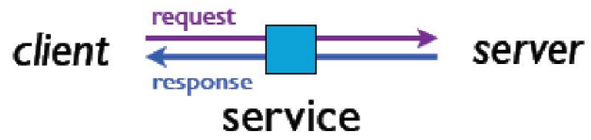

- Messages: **Nodes communicate with each other by passing messages.**
- A message is simply a data structure of typed fields.
- Standard primitive types (integer, floating point, boolean, etc.) are supported, as are arrays of primitive types.
- Messages can include arbitrarily nested structures and arrays (much like C structs).

# Topics



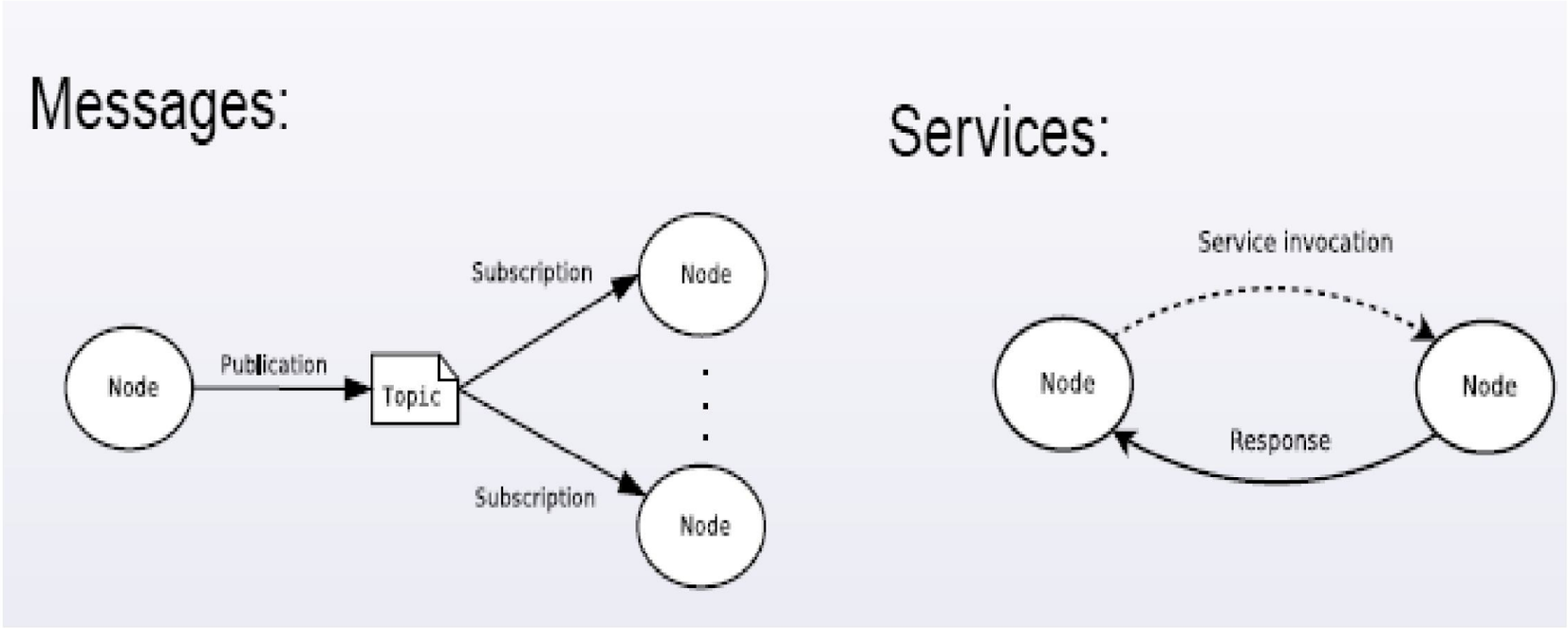

- A node **sends out a message by publishing it to a given topic.**
- The topic is a name that is used to identify the content of the message.
- A node that is interested in a certain kind of data will subscribe to the appropriate topic.
- Logically, one can think of a topic as a strongly typed message bus. Each bus has a name, and anyone can connect to the bus to send or receive messages as long as they are the right type.

# Services




- Publish / subscribe model: many-to-many (messages)
- Request / reply: services
- Uses pair of message structures: one for the request and one for the reply.
- A providing node **offers a service under a name and a client uses the service by sending the request message and awaiting the reply.**

# Messages vs. Services




| Message | Services |
| --- | --- |
| 1:n Communication | 1:1 Communication |
| Non-blocking | blocking |

# Navigating ROS Filesystem

- Filesystem Tools :
- rospack  retrieving information about ROS packages available on the filesystem.
  - Rospack find roscpp
- roscd allows you to change directory directly to a package or a stack.
  - roscd roscpp
  - roscd roscpp/include
  - roscd log
  - roscd (go to the path defined in $ROS_ROOT)
- ROS commands provide Tab-completition
  - roscd tur *TAB key*
- rosls: list directly in a package, stack, or common location by name.
  - rosls roscpp_tutorials
  - rosls roscpp_tutorials/srv

# Creating a ROS Package

- roscreate-pkg: creates a new ROS package.
- Files of a package: manifests, CMakeLists.txt, mainpage.dox, and Makefiles.
  - roscreate-pkg [package_name]
- Dependencies to other packages:
  - roscreate-pkg [package_name] [depend1] [depend2] [depend3]
- Create a package:
  - roscreate-pkg beginner_tutorials std_msgs rospy roscpp
- Update the ROS_PACKAGE_PATH environment variable:

```
export ROS_PACKAGE_PATH=YOUR_PCK_PATH:
$ROS_PACKAGE_PATH
```

# Inspecting the new ROS Package

- Let's take a look at the package: ls
- Content:
  - include/: C++ include headers
  - src/: source files
  - CMakeLists.txt: CMake build file
  - Makefile: the makefile
  - mainpage.dox: Doxygen mainpage documentation
  - manifest.xml: Package manifest file
- Content (not there yet):
  - bin/: compiled binaries
  - msg/: Message (msg) types
  - srv/: Service (srv) types
  - scripts/: executable scripts
- Take a look at the manifest.xml:
  - cat manifest.xml

# ROS Package Dependencies

- What to do if system dependencies are missing?
- rosdep: Resolve system dependencies
- roscd turtlesim
- cat manifest.xml
- Here is a system dep: <rosdep name="wxwidgets"/>
- Install the system dependencies:
- rosdep install turtlesim

# ROS Package Dependencies/Compilation

- rosmake: similar to make command, but does some special "ROS magic".
  - rosdep install turtle_teleop rxtools
- Multiple targets:
  - rosmake turtle_teleop roscpp_tutorials rospy_tutorials rxtools

# Summary Commands

- rospack = ros+pack(age) : provides information related to ROS packages
- rosstack = ros+stack : provides information related to ROS stacks
- roscd = ros+cd : changes directory to a ROS package or stack
- rosls = ros+ls : lists files in a ROS package
- rosdep = ros+dep(endencies) : resolve and install ROS dependencies
- rosmake = ros+make : makes (compiles) a ROS package

# Nodes

- A node really isn't much more than an executable file within a ROS package.
- ROS client libraries allow nodes written in different programming languages to communicate:
  - rospy = python client library
  - roscpp = c++ client library
- rosout: ROS equivalent of stdout/stderr
- **roscore: Master + rosout + parameter server**
- rosnode list displays information about the ROS nodes that are currently running
  - rosnode info : information about a specific node.

# rosrun

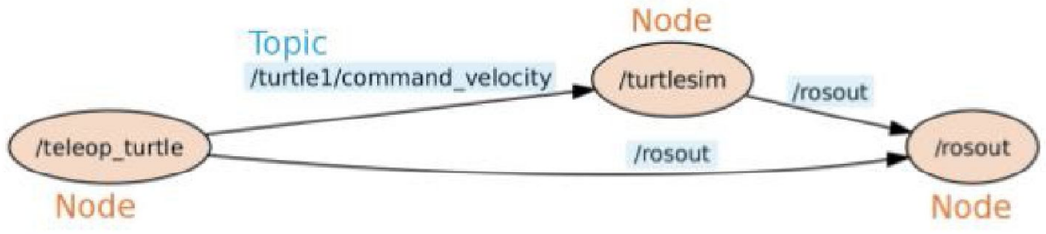
- roscore node has to be running before running any other node
  - roscore
- rosrun allows you to use the package name to directly run a node within a package
  - rosrun [package_name] [node_name]
  - rosrun turtlesim turtlesim_node
  - change the node's name:
  - rosrun turtlesim turtlesim_node __name:=my_turtle

# Review Nodes

- roscore = ros+core : starts the ROS system (master +rosout + parameter server)
- rosnode = ros+node : ROS tool to get information about a node.
- rosrun = ros+run : runs a node from a given package.

# ROS Topics: turtlesim

- Start turtle keyboard teleoperation (in a new terminal):
- rosrun turtlesim turtle_teleop_key
- rxgraph creates a dynamic graph of what's going on in the system.
- turtlesim_node and the turtle_teleop_key: communication over a **ROS Topic.**
- rostopic tool allows you to get information about ROS topics.
- rostopic list




- rostopic echo /turtle1/comand_velocity

# ROS Messages

- Communication on topics happens by sending ROS messages between nodes.
- The publisher and subscriber must send and receive the same type of message.
- The topic **type is defined by the message type published** on it.
  - rostopic type /turtle1/command_velocity
  - rosmsg show turtlesim/Velocity

# Publish Messages

- rostopic pub publishes data on to a topic currently advertised.
  - rostopic pub [topic] [msg_type] [args]
  - rostopic pub -1 /turtle1/command_velocity
    turtlesim/Velocity -- 2.0 1.8
    Observation: Turtle stops moving after some time.
  - Publish steady stream of commands using pub –r command (and removing the -1):
  - rostopic pub /turtle1/command_velocity turtlesim/Velocity -r 1 -- 2.0 1.8

# Services

- Services are another way for nodes to communicate with each other
- Services allow nodes to send a **request and receive a response**
- rosservice tool for listing and querying ROS services:

rosservice list

- rosservice call: Call a service
- Usage: rosservice call [service] [args]
  - rosservice call clear

# roslaunch

- Often we need several nodes
- Typing rosrun for each node is tedious
- roslaunch: starts nodes as defined in a launch file Usage: roslaunch [package] [filename.launch]

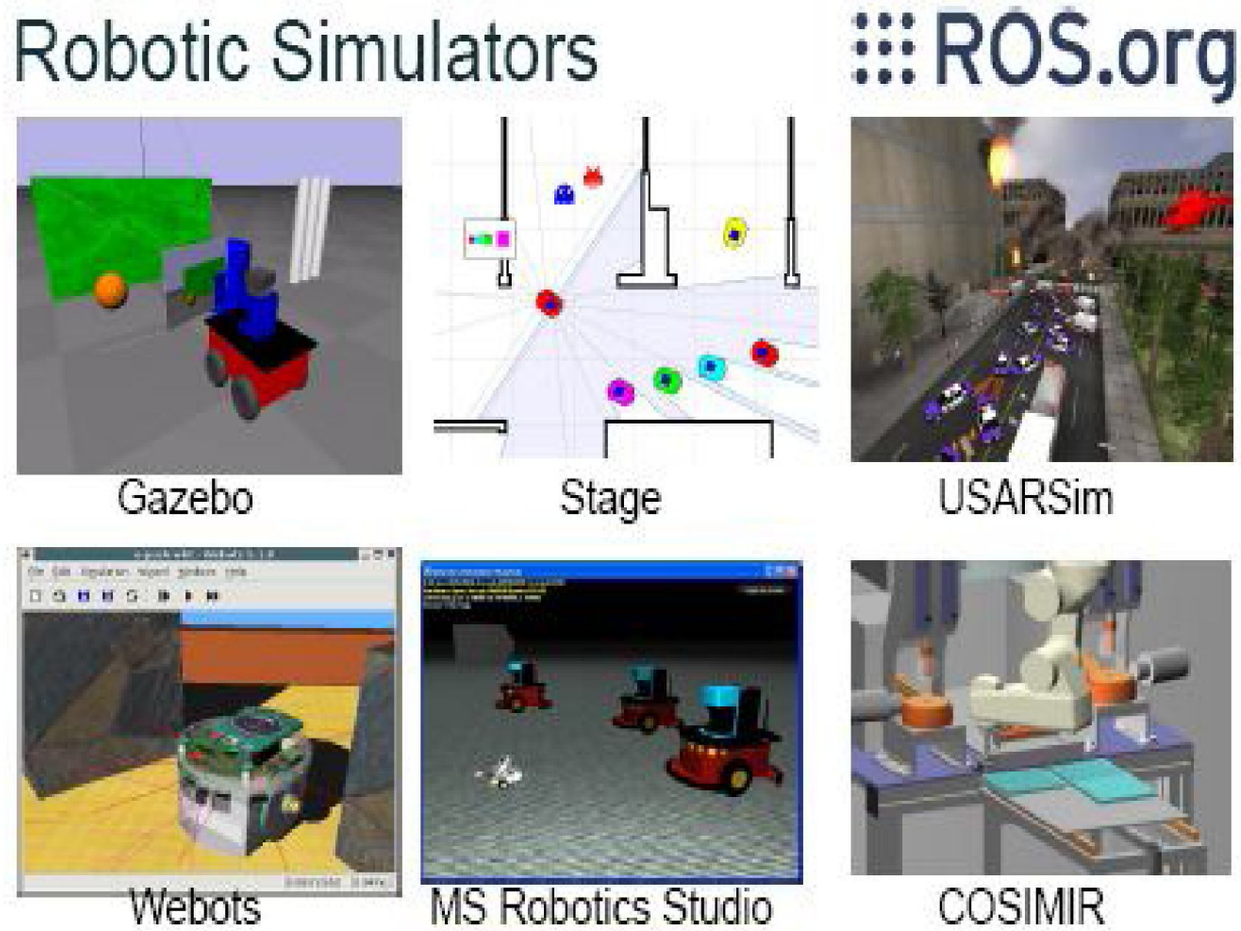
```
<launch>
<group ns="turtlesim1">
<node pkg="turtlesim" name="sim" type="turtlesim_node"/>
</group>
<group ns="turtlesim2">
<node pkg="turtlesim" name="sim" type="turtlesim_node"/>
</group>
<node pkg="turtlesim" name="mimic" type="mimic">
<remap from="input" to="turtlesim1/turtle1"/>
<remap from="output" to="turtlesim2/turtle1"/>
</node>
</launch>
```

# Why Simulators?

- A **robotics simulator is used to create embedded applications for a robot** without depending physically on the actual machine.
- **Cost**
  - No cost for any robot or equipment
  - No risk or damage, no hardware consumption, no maintenance
  - No human risk
- **Time**
  - Simulations can be run in parallel
  - No battery recharge
- **Experiments**
  - Any environment, any robot, any sensor
  - Experimental repeatability
  - Scalability

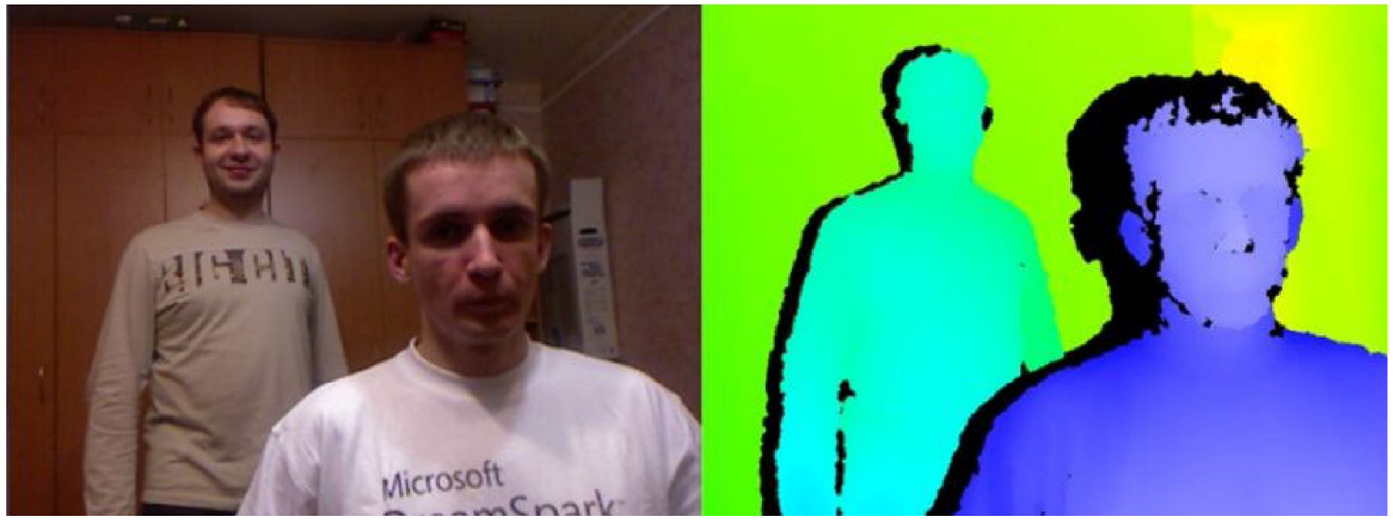
# Stage/Gazebo

- **Stage**
- Scalable 2D multiple robot simulator
- Light simulation
- **Gazebo**
- High-fidelity 3D multiple robot simulator
- Open Dynamics Engine



# Openni Kinect


- New sensor developed by PrimeSense and Microsoft
- ROS OpenNI is an open source project focused on the integration of the PrimeSense sensors with ROS.
- sudo apt-get install ros-electric-openni-kinect
- A ROS driver for OpenNI depth (+ RGB) cameras. These include:
- roslaunch openni_launch openni.launch
- rosrun image_view disparity_view image:=/camera/depth/disparity
- rosrun image_view image_view image:=/camera/rgb/image_color

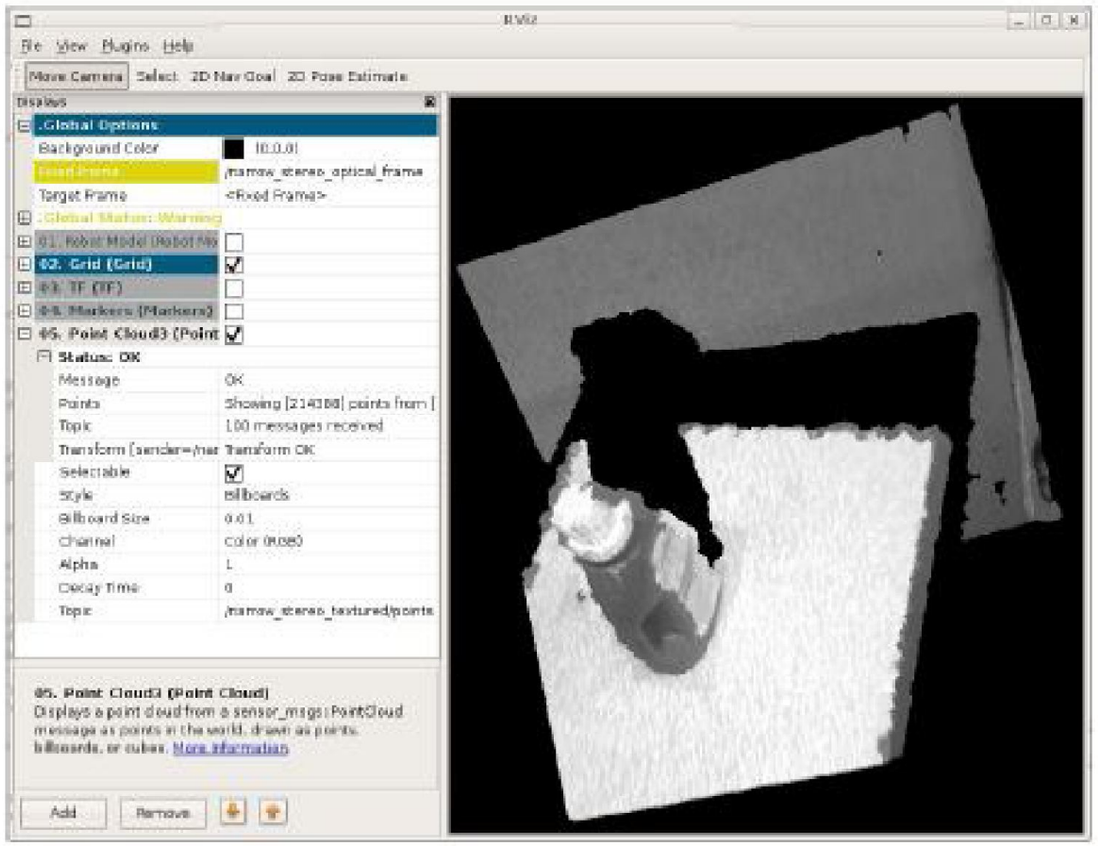


# Viewing Point Clouds

- In Rviz (A 3d visualization environment for robots using ROS)
  - Rosrun rviz rviz

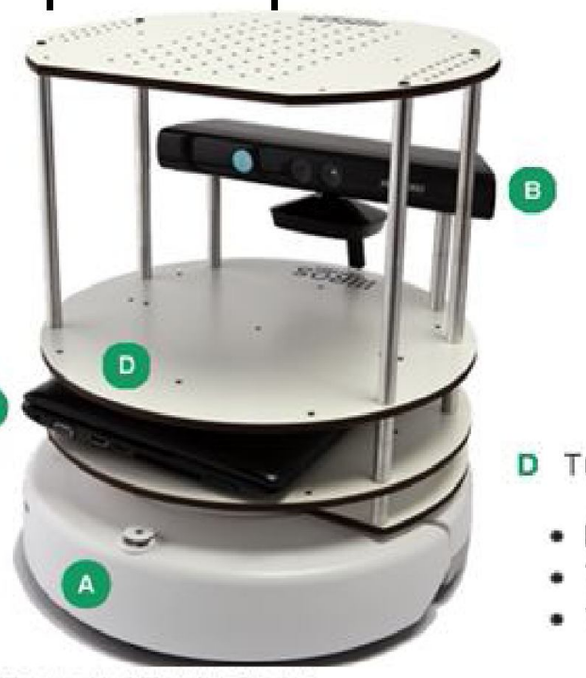
Displays panel → Add → Point Cloud

Enter the topic in the red box



# TurtleBot

- TurtleBot combines popular off-the-shelf robot components like the iRobot Create and Kinect into an integrated development platform for ROS applications




**B** 3D Sensor

- Microsoft Kinect
- Kinect Power Board Adapter Cable

**C** Computing :: ASUS

**D** TurtleBot Hardware

- Kinect Mounting Hardware
- TurtleBot Structure
- TurtleBot Module Plate with 1 inch Spacing Hole Pattern

**A** Mobile Base and Power Board

- iRobot Create
- 3000 mAh Ni-MH Battery Pack
- 150 degrees/second Single Axis Gyro
- 12V 1.5Amp Software Enabled Power Supply (for powering the Kinect)

# Turtlebot

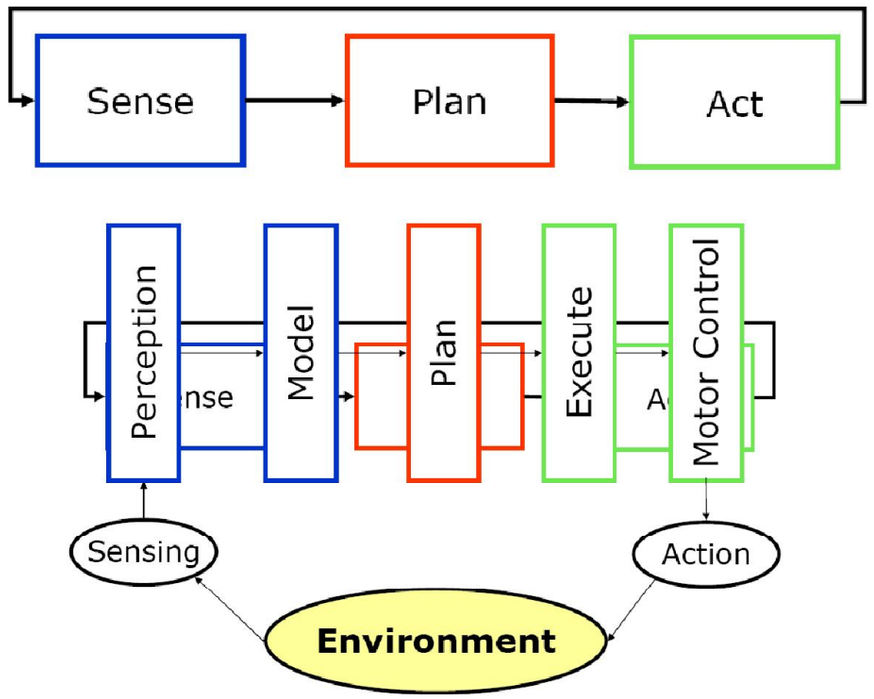



Figure: Functional decomposition

Video: Turtlebot introduction (courtesy: ROS wiki)

# Communication Layout

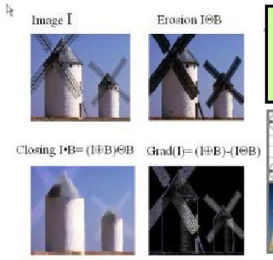
- When working with the turtlebot there are two computers involved.
  - First one, the **netbook**, is the laptop attached to the physical turtlebot.
  - Second one, the **workstation** PC to remotely work on the turtlebot. Students should always use the workstation to interact with the turtlebot. Netbook should be safely fitted to the turtlebot at all times.

# OpenCV: (1999, from 2009 WillowGarage)

**> 2000 algorithms**

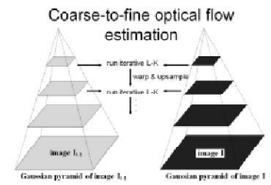
OpenCV

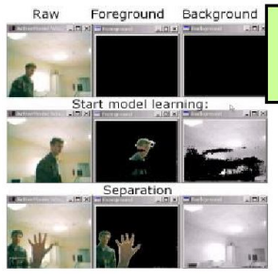
opencv.willowgarage.con




## General Image Processing Functions

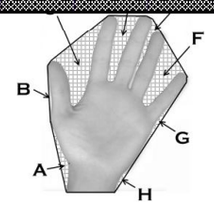
## Image Pyramids

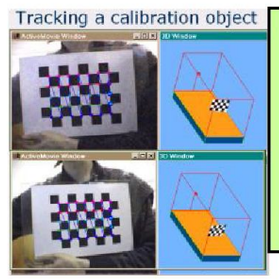


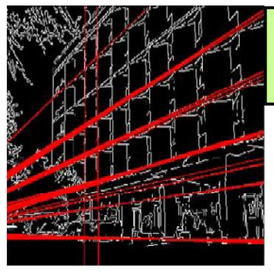

## Segmentation

## Geometric descriptors

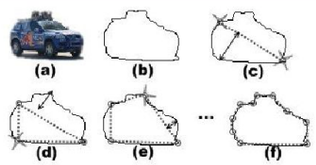


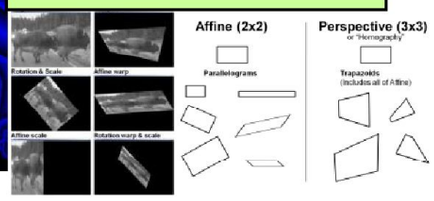

## Camera calibration, Stereo, 3D

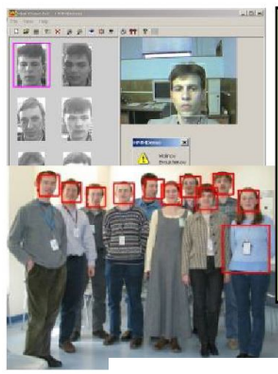


## Features




## Transforms

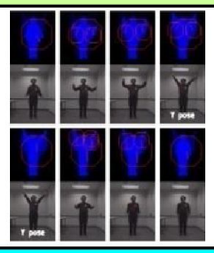


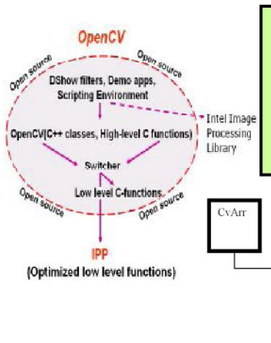
## Machine Learning:

- Detection,
- Recognition

## Tracking

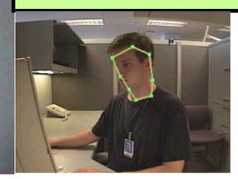



## Matrix Math

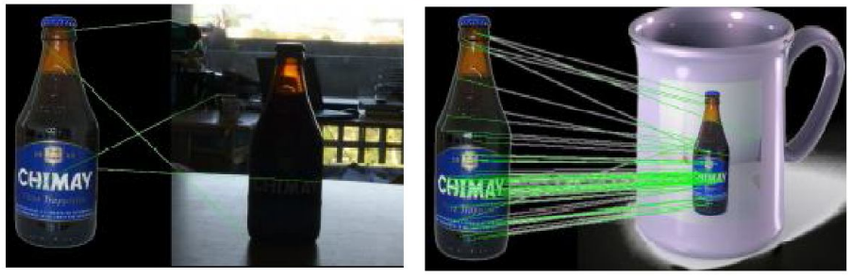



## Utilities and Data Structures

## Fitting

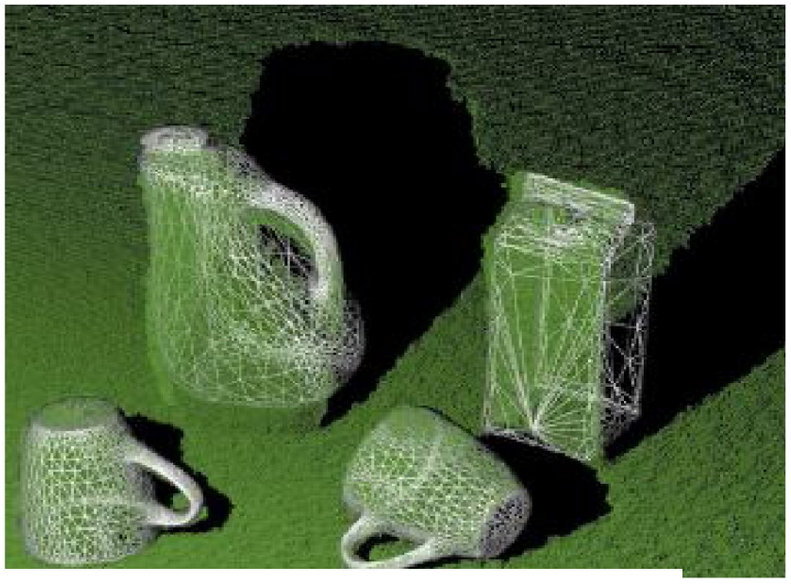


# 3D Processing

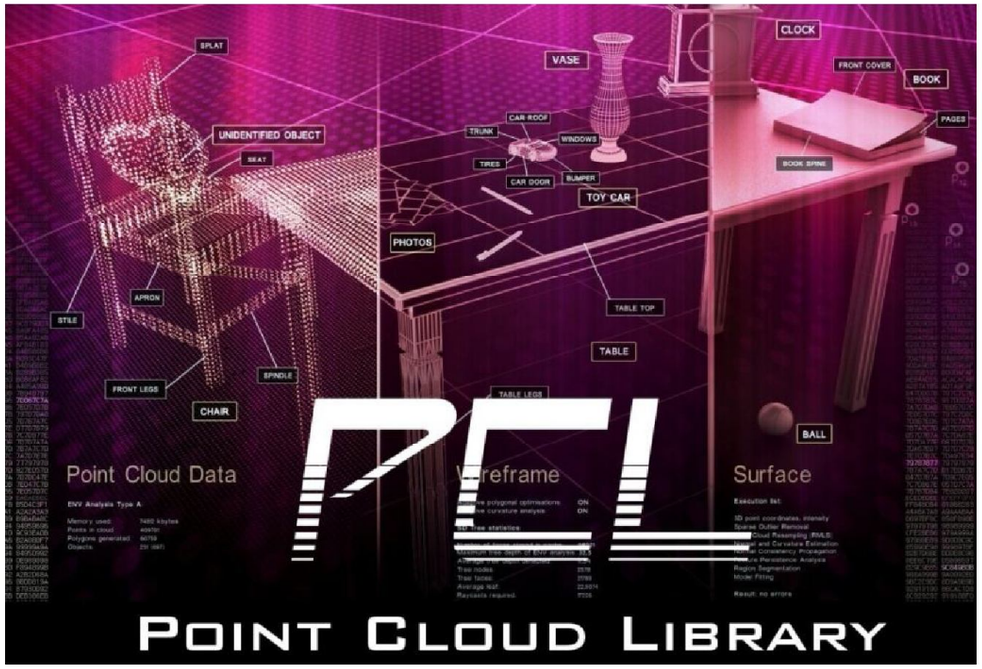



- The world is not 2D
- 2D only vision might give erroneous results
- 3D has arrived!
- Kinect, 3D pocket cameras!


- n-D Perception:
- at the very least n = 3: x,y,z.
- + RGB = 4D
- + normal information = 7D
- + surface curvature = 8D
- + ... = n-D



# Point Cloud Library (PCL)




- 3D features
- Segmentation
- Surface reconstruction
- Registration
- Keypoints

## Collection of smaller, modular libraries:

**libpcl_features**: 3D features (e.g., normals and curvatures, boundary points, moment invariants, principal curvatures, Point Feature Histograms (PFH), Fast PFH, NARF, RSD...)
**libpcl_surface**: surface reconstruction (e.g., meshing, convex hulls, Moving Least Squares, …)
**libpcl_filters**: point cloud filters (e.g., down sampling, outlier removal, indices extraction, projections, ...)
**libpcl_segmentation**: segmentation (e.g.,cluster extraction, Sample Consensus model fitting, polygonal prism extraction, …)
**libpcl_registration**: alignment methods (e.g., Iterative Closest Point, non linear optimizations, …)
**libpcl_io**: I/O operations

# Example: PCL Segmentation