Back (/tutorials?cat=)

Intermediate: Control plugin

Recap

At this point we have an almost fully functional sensor. The model components are in place, it's been added to Gazebo's online database, and a Gaussian noise model has been applied. The final component to add is a plugin that controls the sensor's one degree of freedom. If you skipped the previous tutorials then download the model here. (https://github.com/osrf/gazebo_tutorials/raw/master/guided_i/files/velodyne_hdl32.tar.gz)

Plugin overview

A plugin is a C++ library that is loaded by Gazebo at runtime. A plugin has access to Gazebo's API, which allows a plugin to perform a wide variety of tasks including moving objects, adding/removing objects, and accessing sensor data.

More information on plugins is available in these tutorials (http://gazebosim.org/tutorials?cat=write_plugin). It is highly recommended that you look over these tutorials before proceeding.

Install Gazebo headers

On some operating systems the development package must be installed prior to building a plugin.

```
# Ubuntu or Debian
sudo apt install libgazebo8-dev
# Fedora
sudo yum install gazebo-devel
```

Write the plugin

We will create the plugin in a new directory. The contents of this directory will include the plugin source code, and a CMake build script.

We won't go into heavy detail about the various components of a plugin in order to keep this tutorial fairly short. Take a look at these other tutorials (http://gazebosim.org/tutorials?cat=write_plugin) for more information.

Step 1: Create a workspace

```
mkdir ~/velodyne_plugin
cd ~/velodyne_plugin
```

Step 2: Create the plugin source file

We will start with a bare bones plugin, just to get things rolling.

Inside your workspace, create the source file.

```
gedit velodyne_plugin.cc
```

Copy in the following code.

```
#ifndef _VELODYNE_PLUGIN_HH_
#define _VELODYNE_PLUGIN_HH_

#include <gazebo/gazebo.hh>
#include <gazebo/physics/physics.hh>

namespace gazebo
{
  /// \brief A plugin to control a Velodyne sensor.
  class VelodynePlugin : public ModelPlugin
  {
    /// \brief Constructor
    public: VelodynePlugin() {}

    /// \brief The load function is called by Gazebo when the plugin is
    /// inserted into simulation
    /// \param[in] _model A pointer to the model that this plugin is
    /// attached to.
    /// \param[in] _sdf A pointer to the plugin's SDF element.
    public: virtual void Load(physics::ModelPtr _model, sdf::ElementPtr _sdf)
    {
      // Just output a message for now
      std::cerr << "\nThe velodyne plugin is attach to model[" <<
        _model->GetName() << "]\n";
    }
  };

  // Tell Gazebo about this plugin, so that Gazebo can call Load on this plugin.
  GZ_REGISTER_MODEL_PLUGIN(VelodynePlugin)
}
#endif
```

Step 3: Create the CMake build script

Create a `CMakeLists.txt` file inside your workspace.

```
gedit CMakeLists.txt
```

Copy in the following content.

```
cmake_minimum_required(VERSION 2.8 FATAL_ERROR)

# Find Gazebo
find_package(gazebo REQUIRED)
include_directories(${GAZEBO_INCLUDE_DIRS})
link_directories(${GAZEBO_LIBRARY_DIRS})
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${GAZEBO_CXX_FLAGS}")

# Build our plugin
add_library(velodyne_plugin SHARED velodyne_plugin.cc)
target_link_libraries(velodyne_plugin ${GAZEBO_LIBRARIES})
```

Step 4: Attach the plugin to the Velodyne sensor

We will utilize SDF's `<include>` capability to test out our plugin without touching the main Velodyne SDF file.

Inside your workspace, create a new world file.

```
gedit velodyne.world
```

Copy in the following SDF content.

```
<?xml version="1.0" ?>
<sdf version="1.5">
  <world name="default">
    <!-- A global light source -->
    <include>
      <uri>model://sun</uri>
    </include>
    <!-- A ground plane -->
    <include>
      <uri>model://ground_plane</uri>
    </include>

    <!-- A testing model that includes the Velodyne sensor model -->
    <model name="my_velodyne">
      <include>
        <uri>model://velodyne_hdl32</uri>
      </include>

      <!-- Attach the plugin to this model -->
      <plugin name="velodyne_control" filename="libvelodyne_plugin.so"/>
    </model>

  </world>
</sdf>
```

Step 5: Build and test

We are now ready to compile the plugin, and test it out.

Within your workspace, create a build directory.

```
mkdir build
```

Compile the plugin.

```
cd build
cmake ..
make
```

Run the world. Note: It is important to run gazebo from within the `build` directory so that Gazebo can find the plugin library.

1. Gazebo version < 6

```
cd ~/velodyne_plugin/build
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:~/velodyne_plugin/build
gazebo ../velodyne.world
```

2. Gazebo version >= 6

```
cd ~/velodyne_plugin/build
gazebo --verbose ../velodyne.world
```

Check your terminal, you should see:

```
The velodyne plugin is attached to model[my_velodyne]
```

Move the Velodyne

We now have a basic plugin that can be compiled and run by Gazebo. The next step is to add code that controls the Velodyne's joint.

We will use a simple PID controller to control the velocity of the Velodyne's joint.

Open the source file in your workspace.

```
gedit ~/velodyne_plugin/velodyne_plugin.cc
```

Modify the **`Load`** function to have the following content.

```
public: virtual void Load(physics::ModelPtr _model, sdf::ElementPtr _sdf)
{
  // Safety check
  if (_model->GetJointCount() == 0)
  {
    std::cerr << "Invalid joint count, Velodyne plugin not loaded\n";
    return;
  }

  // Store the model pointer for convenience.
  this->model = _model;

  // Get the first joint. We are making an assumption about the model
  // having one joint that is the rotational joint.
  this->joint = _model->GetJoints()[0];

  // Setup a P-controller, with a gain of 0.1.
  this->pid = common::PID(0.1, 0, 0);

  // Apply the P-controller to the joint.
  this->model->GetJointController()->SetVelocityPID(
      this->joint->GetScopedName(), this->pid);

  // Set the joint's target velocity. This target velocity is just
  // for demonstration purposes.
  this->model->GetJointController()->SetVelocityTarget(
      this->joint->GetScopedName(), 10.0);
}
```

And add the following private members to the class, just below the `Load` function:

```
/// \brief Pointer to the model.
private: physics::ModelPtr model;

/// \brief Pointer to the joint.
private: physics::JointPtr joint;

/// \brief A PID controller for the joint.
private: common::PID pid;
```

Recompile and run Gazebo.

```
cd ~/velodyne_plugin/build
make
gazebo --verbose ../velodyne.world
```

You should see the Velodyne spinning.

Plugin Configuration

We have hardcoded the rotational speed, however a configurable speed that doesn't require re-compilation would be better. In this section we'll modify the plugin to read a custom SDF parameter that is the target velocity of the Velodyne.

Start by adding a new element as a child of the `<plugin>` . The new element can be anything, as long as it is valid XML. Our plugin will have access to this value in the `Load` function.

```
gedit ~/velodyne_plugin/velodyne.world
```

Modify the `<plugin>` to contain a new `<velocity>` element.

```
<plugin name="velodyne_control" filename="libvelodyne_plugin.so">
  <velocity>25</velocity>
</plugin>
```

Now let's read this value in the plugin's `Load` function.

```
gedit ~/velodyne_plugin/velodyne_plugin.cc
```

Modify the end of the `Load` function to read the `<velocity>` using the `sdf::ElementPtr` parameter.

```
// Default to zero velocity
double velocity = 0;

// Check that the velocity element exists, then read the value
if (_sdf->HasElement("velocity"))
  velocity = _sdf->Get<double>("velocity");

// Set the joint's target velocity. This target velocity is just
// for demonstration purposes.
this->model->GetJointController()->SetVelocityTarget(
    this->joint->GetScopedName(), velocity);
```

Compile and run simulation to see the results.

```
cd ~/velodyne_plugin/build
cmake ../
make
gazebo --verbose ../velodyne.world
```

Adjust the `<velocity>` SDF value, and restart simulation to see the effects.

Create an API

Adjusting the target velocity via SDF is very convenient, but it would be even better to support dynamic adjustments. This change will require the addition of an API that other programs can use to change the velocity value.

There are two API types that we can use: message passing, and functions. Message passing relies on Gazebo's transport mechanism, and would involve creating a named topic on which a publisher can send double values. The plugin would receive these messages, containing a `double` , and set the velocity appropriately. Message passing is convenient for inter-process communication.

The function approach would create a new public function that adjusts the velocity. For this to work, a new plugin would inherit from our current plugin. The child plugin would be instantiated by Gazebo instead of our current plugin, and would control the velocity by calling our function. This type of approach is most often used when interfacing Gazebo to ROS.

Due to the simplicity of our plugin, it's easy to implement both simultaneously.

1. Start by opening the `velodyne_plugin.cc` file.

```
gedit ~/velodyne_plugin/velodyne_plugin.cc
```

2. Create a new public function that can set the target velocity. This will fullfill the functional API.

```
/// \brief Set the velocity of the Velodyne
/// \param[in] _vel New target velocity
public: void SetVelocity(const double &_vel)
{
  // Set the joint's target velocity.
  this->model->GetJointController()->SetVelocityTarget(
      this->joint->GetScopedName(), _vel);
}
```

3. Now we will setup the messaging passing infrastructure.

    1. Add a Node and subscriber to the plugin.

    ```
    /// \brief A node used for transport
    private: transport::NodePtr node;

    /// \brief A subscriber to a named topic.
    private: transport::SubscriberPtr sub;
    ```

    2. Instantiate the Node and subscriber at the end of `Load` function.

    ```
    // Create the node
    this->node = transport::NodePtr(new transport::Node());
    #if GAZEBO_MAJOR_VERSION < 8
    this->node->Init(this->model->GetWorld()->GetName());
    #else
    this->node->Init(this->model->GetWorld()->Name());
    #endif

    // Create a topic name
    std::string topicName = "~/" + this->model->GetName() + "/vel_cmd";

    // Subscribe to the topic, and register a callback
    this->sub = this->node->Subscribe(topicName,
       &VelodynePlugin::OnMsg, this);
    ```

    3. Create the callback function that handles incoming messages

    ```
    /// \brief Handle incoming message
    /// \param[in] _msg Repurpose a vector3 message. This function will
    /// only use the x component.
    private: void OnMsg(ConstVector3dPtr &_msg)
    {
      this->SetVelocity(_msg->x());
    }
    ```

    4. Add two necessary headers to the plugin.

```
#include <gazebo/transport/transport.hh>
#include <gazebo/msgs/msgs.hh>
```

4. The plugin is now ready to dynamically alter the target velocity. The complete plugin should look like this.

```
#ifndef _VELODYNE_PLUGIN_HH_
#define _VELODYNE_PLUGIN_HH_

#include <gazebo/gazebo.hh>
#include <gazebo/physics/physics.hh>
#include <gazebo/transport/transport.hh>
#include <gazebo/msgs/msgs.hh>

namespace gazebo
{
  /// \brief A plugin to control a Velodyne sensor.
  class VelodynePlugin : public ModelPlugin
  {
    /// \brief Constructor
    public: VelodynePlugin() {}

    /// \brief The load function is called by Gazebo when the plugin is
    /// inserted into simulation
    /// \param[in] _model A pointer to the model that this plugin is
    /// attached to.
    /// \param[in] _sdf A pointer to the plugin's SDF element.
    public: virtual void Load(physics::ModelPtr _model, sdf::ElementPtr _sdf)
    {
      // Safety check
      if (_model->GetJointCount() == 0)
      {
        std::cerr << "Invalid joint count, Velodyne plugin not loaded\n";
        return;
      }

      // Store the model pointer for convenience.
      this->model = _model;

      // Get the first joint. We are making an assumption about the model
      // having one joint that is the rotational joint.
      this->joint = _model->GetJoints()[0];

      // Setup a P-controller, with a gain of 0.1.
      this->pid = common::PID(0.1, 0, 0);

      // Apply the P-controller to the joint.
      this->model->GetJointController()->SetVelocityPID(
          this->joint->GetScopedName(), this->pid);

      // Default to zero velocity
      double velocity = 0;

      // Check that the velocity element exists, then read the value
      if (_sdf->HasElement("velocity"))
        velocity = _sdf->Get<double>("velocity");

      this->SetVelocity(velocity);

      // Create the node
      this->node = transport::NodePtr(new transport::Node());
      #if GAZEBO_MAJOR_VERSION < 8
      this->node->Init(this->model->GetWorld()->GetName());
```

```
      #else
      this->node->Init(this->model->GetWorld()->Name());
      #endif

      // Create a topic name
      std::string topicName = "~/" + this->model->GetName() + "/vel_cmd";

      // Subscribe to the topic, and register a callback
      this->sub = this->node->Subscribe(topicName,
         &VelodynePlugin::OnMsg, this);
    }

    /// \brief Set the velocity of the Velodyne
    /// \param[in] _vel New target velocity
    public: void SetVelocity(const double &_vel)
    {
      // Set the joint's target velocity.
      this->model->GetJointController()->SetVelocityTarget(
          this->joint->GetScopedName(), _vel);
    }

    /// \brief Handle incoming message
    /// \param[in] _msg Repurpose a vector3 message. This function will
    /// only use the x component.
    private: void OnMsg(ConstVector3dPtr &_msg)
    {
      this->SetVelocity(_msg->x());
    }

    /// \brief A node used for transport
    private: transport::NodePtr node;

    /// \brief A subscriber to a named topic.
    private: transport::SubscriberPtr sub;

    /// \brief Pointer to the model.
    private: physics::ModelPtr model;

    /// \brief Pointer to the joint.
    private: physics::JointPtr joint;

    /// \brief A PID controller for the joint.
    private: common::PID pid;
  };

  // Tell Gazebo about this plugin, so that Gazebo can call Load on this plugin.
  GZ_REGISTER_MODEL_PLUGIN(VelodynePlugin)
}
#endif
```

Test the message passing API

We will test the API with a new program that publishes messages to the plugin.

1. Create a new source file in your workspace.

```
gedit ~/velodyne_plugin/vel.cc
```

2. Add the following code. Comments in the code explain what is going on.

```
#include <gazebo/gazebo_config.h>
#include <gazebo/transport/transport.hh>
#include <gazebo/msgs/msgs.hh>

// Gazebo's API has changed between major releases. These changes are
// accounted for with #if..#endif blocks in this file.
#if GAZEBO_MAJOR_VERSION < 6
#include <gazebo/gazebo.hh>
#else
#include <gazebo/gazebo_client.hh>
#endif

/////////////////////////////////////////////////
int main(int _argc, char **_argv)
{
  // Load gazebo as a client
#if GAZEBO_MAJOR_VERSION < 6
  gazebo::setupClient(_argc, _argv);
#else
  gazebo::client::setup(_argc, _argv);
#endif

  // Create our node for communication
  gazebo::transport::NodePtr node(new gazebo::transport::Node());
  node->Init();

  // Publish to the  velodyne topic
  gazebo::transport::PublisherPtr pub =
    node->Advertise<gazebo::msgs::Vector3d>("~/my_velodyne/vel_cmd");

  // Wait for a subscriber to connect to this publisher
  pub->WaitForConnection();

  // Create a a vector3 message
  gazebo::msgs::Vector3d msg;

  // Set the velocity in the x-component
#if GAZEBO_MAJOR_VERSION < 6
  gazebo::msgs::Set(&msg, gazebo::math::Vector3(std::atof(_argv[1]), 0, 0));
#else
  gazebo::msgs::Set(&msg, ignition::math::Vector3d(std::atof(_argv[1]), 0, 0));
#endif

  // Send the message
  pub->Publish(msg);

  // Make sure to shut everything down.
#if GAZEBO_MAJOR_VERSION < 6
  gazebo::shutdown();
#else
  gazebo::client::shutdown();
#endif
}
```

3. Add a few lines to the `CMakeLists.txt` file in your workspace, to build the new `vel` program.

```
gedit ~/velodyne_plugin/CMakeLists.txt
```

```
# Build the stand-alone test program
add_executable(vel vel.cc)

if (${gazebo_VERSION_MAJOR} LESS 6)
  # These two
  include(FindBoost)
  find_package(Boost ${MIN_BOOST_VERSION} REQUIRED system filesystem regex)
  target_link_libraries(vel ${GAZEBO_LIBRARIES} ${Boost_LIBRARIES})
else()
  target_link_libraries(vel ${GAZEBO_LIBRARIES})
endif()
```

4. Compile and run simulation

```
cd ~/velodyne_plugin/build
cmake ../
make
gazebo --verbose ../velodyne.world
```

5. In a new terminal, go into the build directory and run the `vel` command. Make sure to set number value, which is interpreted as the target velocity value.

```
cd ~/velodyne_plugin/build
./vel 2
```

6. You can now dynamically set the velocity of the Velodyne senor.

Next up

The final tutorial in this series discusses how to connect this plugin to ROS.

Connect to ROS (http://gazebosim.org/tutorials?cat=guided_i&tut=guided_i6)