Visualizing In RViz

moveit.picknik.ai/humble/doc/tutorials/visualizing_in_rviz/visualizing_in_rviz.html

This tutorial will introduce you to a tool that can help you more easily understand what your MoveIt application is doing by rendering visualizations in RViz.

Prerequisites

If you haven't already done so, make sure you've completed the steps in Your First Project. This project assumes you are starting with the `hello_moveit` project, where the previous tutorial left off.

Steps

1 Add the dependency moveit_visual_tools_

Add this line to your `package.xml` in the `hello_moveit` project after the other `<depend>` statements:

```
<depend>moveit_visual_tools</depend>
```

Then in your `CMakeLists.txt` add this line to the section of `find_package` statements:

```
find_package(moveit_visual_tools REQUIRED)
```

Further down in the file extend the `ament_target_dependencies` macro call to include the new dependency like this:

```
ament_target_dependencies(
  hello_moveit
  "moveit_ros_planning_interface"
  "moveit_visual_tools"
  "rclcpp"
)
```

i feel this really really complex and redundant

To verify that you added the dependency correctly, add the required include to your source file `hello_moveit.cpp` :

```
#include <moveit_visual_tools/moveit_visual_tools.h>
```

To test that this all worked, open a terminal in the workspace directory (remembering to source your ROS install in opt) and then build with colcon:

```
cd ~/ws_moveit2
colcon build --mixin debug
```

always.....

2 Create a ROS executor and spin the node on a thread_

Before we can initialize MoveItVisualTools, we need to have a executor spinning on our ROS node. This is necessary because of how MoveItVisualTools interacts with ROS services and topics.

在我们能够初始化MoveItVisualTools之前，我们需要在我们的ROS节点上有一个执行器在旋转。这是必要的，因为MoveItVisualTools如何与ROS服务和主题互动。

```
#include <thread>  // <---- add this to the set of includes at the top

  ...

  // Create a ROS logger
  auto const logger = rclcpp::get_logger("hello_moveit");

  // We spin up a SingleThreadedExecutor so MoveItVisualTools interact with ROS
  rclcpp::executors::SingleThreadedExecutor executor;
  executor.add_node(node);
  auto spinner = std::thread([&executor]() { executor.spin(); });

  // Create the MoveIt MoveGroup Interface
  ...

  // Shutdown ROS
  rclcpp::shutdown();  // <--- This will cause the spin function in the thread to return
  spinner.join();  // <--- Join the thread before exiting
  return 0;
}
```

After each one of these changes, you should rebuild your workspace to make sure you don't have any syntax errors.

3 Create and Initialize MoveItVisualTools_

Next, we will construct and initialize MoveItVisualTools after the construction of MoveGroupInterface.

```
// Create the MoveIt MoveGroup Interface
using moveit::planning_interface::MoveGroupInterface;
auto move_group_interface = MoveGroupInterface(node, "panda_arm");

// Construct and initialize MoveItVisualTools
auto moveit_visual_tools = moveit_visual_tools::MoveItVisualTools{
    node, "panda_link0", rviz_visual_tools::RVIZ_MARKER_TOPIC,
    move_group_interface.getRobotModel()};
moveit_visual_tools.deleteAllMarkers();
moveit_visual_tools.loadRemoteControl();
```

We pass the following into the constructor: the ROS node, the base link of the robot, the marker topic to use (more on this later), and the robot model (which we get from the move_group_interface). Next, we make a call to delete all the markers. This clears any rendered state out of RViz that we have left over from previous runs. Lastly, we load remote control. Remote control is a really simple plugin that lets us have a button in RViz to interact with our program.

4 Write closures for visualizations_

After we've constructed and initialized, we now create some closures (function objects that have access to variables in our current scope) that we can use later in our program to help render visualizations in RViz.

```
// Create a closures for visualization
auto const draw_title = [&moveit_visual_tools](auto text) {
  auto const text_pose = [] {
    auto msg = Eigen::Isometry3d::Identity();
    msg.translation().z() = 1.0;
    return msg;
  }();
  moveit_visual_tools.publishText(text_pose, text, rviz_visual_tools::WHITE,
                                  rviz_visual_tools::XLARGE);
};
auto const prompt = [&moveit_visual_tools](auto text) {
  moveit_visual_tools.prompt(text);
};
auto const draw_trajectory_tool_path =
    [&moveit_visual_tools,
     jmg = move_group_interface.getRobotModel()->getJointModelGroup(
         "panda_arm")](auto const trajectory) {
      moveit_visual_tools.publishTrajectoryLine(trajectory, jmg);
    };
```

Each of the three closures capture `moveit_visual_tools` by reference and the last one captures a pointer to the joint model group object we are planning with. Each of these call a function on `moveit_visual_tools` that changes something in RViz. The first one, `draw_title` adds text one meter above the base of the robot. This is a useful way to show the state of your program from a high level. The second one calls a function called `prompt`. This function blocks your program until the user presses the `next` button in RViz. This is helpful for stepping through a program when debugging. The last one draws the tool path of a trajectory that we have planned. This is often helpful for understanding a planned trajectory from the perspective of the tool.

You might be asking yourself why we would create lambdas like this, and the reason is simply to make the code that comes later easier to read and understand. As your write software, it is often helpful to break up your functionality into named functions which can be easily reused and tested on their own. You will see in the next section how we use these functions we created.

5 Visualize the steps of your program_

Now we'll augment the code in the middle of your program. Update your code for planning and executing to include these new features:

这三个闭包中的每一个都通过引用来捕获moveit_visual_tools，最后一个捕获了一个指向我们正在计划的联合模型组对象的指针。每一个都调用了moveit_visual_tools的一个函数，以改变RViz中的一些东西。第一个函数，draw_title在机器人底座上方一米处添加文本。这是一个有用的方法，可以从高处显示你的程序的状态。第二个是调用一个叫做prompt的函数。这个函数阻止你的程序，直到用户在RViz中按下一个按钮。这对于在调试时逐步完成一个程序很有帮助。最后一个是画出我们计划的轨迹的工具路径。这通常有助于从工具的角度理解计划中的轨迹。你可能会问自己，为什么我们要创建这样的lambdas，原因很简单，就是为了让后面的代码更容易阅读和理解。在你写软件的时候，把你的功能分成命名的函数往往是很有帮助的，这些函数可以很容易地被重复使用和单独测试。你将在下一节看到我们如何使用这些我们创建的函数。

```
// Set a target Pose
auto const target_pose = [] {
  geometry_msgs::msg::Pose msg;
  msg.orientation.w = 1.0;
  msg.position.x = 0.28;
  msg.position.y = -0.2;
  msg.position.z = 0.5;
  return msg;
}();
move_group_interface.setPoseTarget(target_pose);

// Create a plan to that target pose
prompt("Press 'Next' in the RvizVisualToolsGui window to plan");
draw_title("Planning");
moveit_visual_tools.trigger();
auto const [success, plan] = [&move_group_interface] {
  moveit::planning_interface::MoveGroupInterface::Plan msg;
  auto const ok = static_cast<bool>(move_group_interface.plan(msg));
  return std::make_pair(ok, msg);
}();

// Execute the plan
if (success) {
  draw_trajectory_tool_path(plan.trajectory_);
  moveit_visual_tools.trigger();
  prompt("Press 'Next' in the RvizVisualToolsGui window to execute");
  draw_title("Executing");
  moveit_visual_tools.trigger();
  move_group_interface.execute(plan);
} else {
  draw_title("Planning Failed!");
  moveit_visual_tools.trigger();
  RCLCPP_ERROR(logger, "Planing failed!");
}
```

One thing you'll quickly notice is that we have to call a method called `trigger` on `moveit_visual_tools` after each call to change something rendered in RViz. The reason for this is that messages sent to RViz are batched up and sent when you call `trigger` to reduce bandwidth of the marker topics.

Lastly, build your project again to make sure all the code additions are correct.

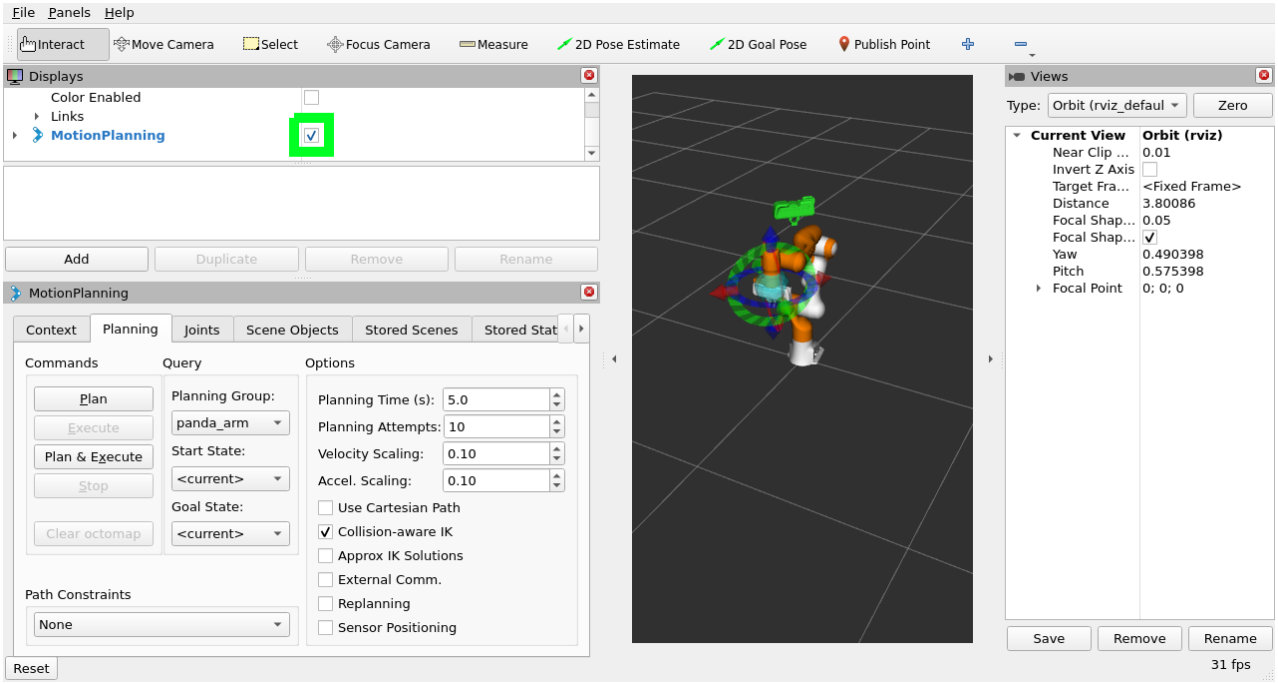
```
cd ~/ws_moveit2
source /opt/ros/humble/setup.bash
colcon build --mixin debug
```

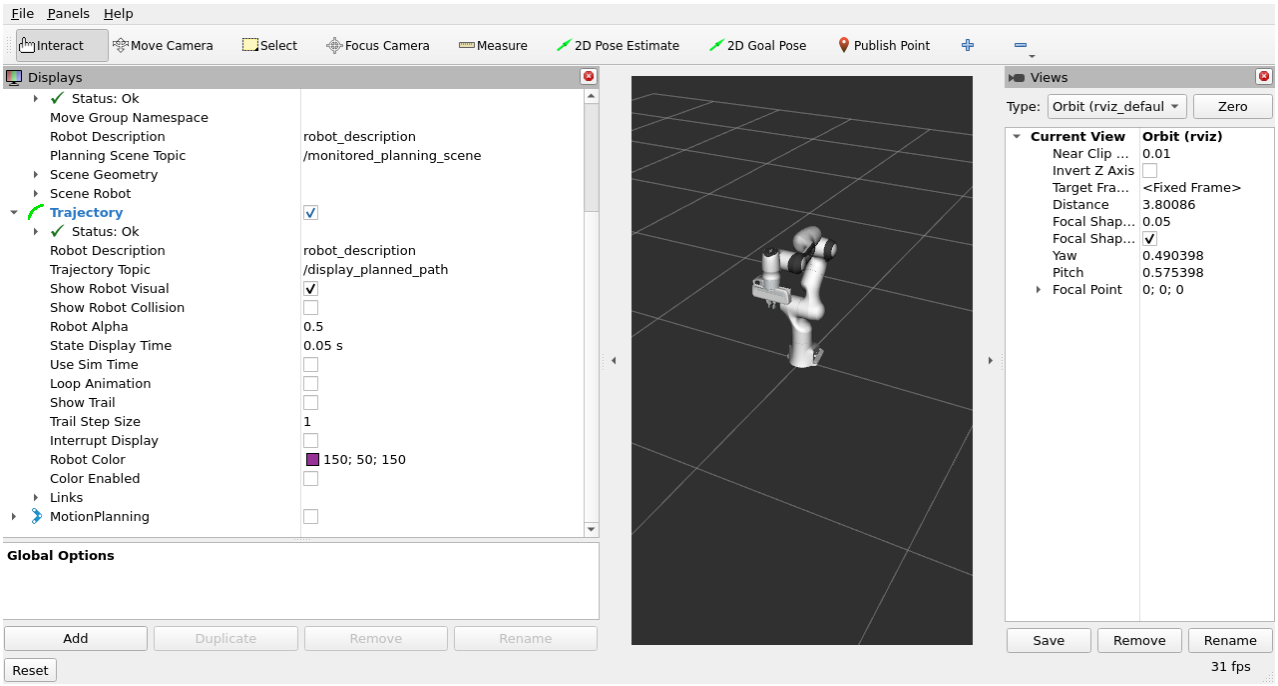
6 Enable visualizations in RViz_

Open a new terminal, source the workspace, and then start the demo launch file that opens RViz.

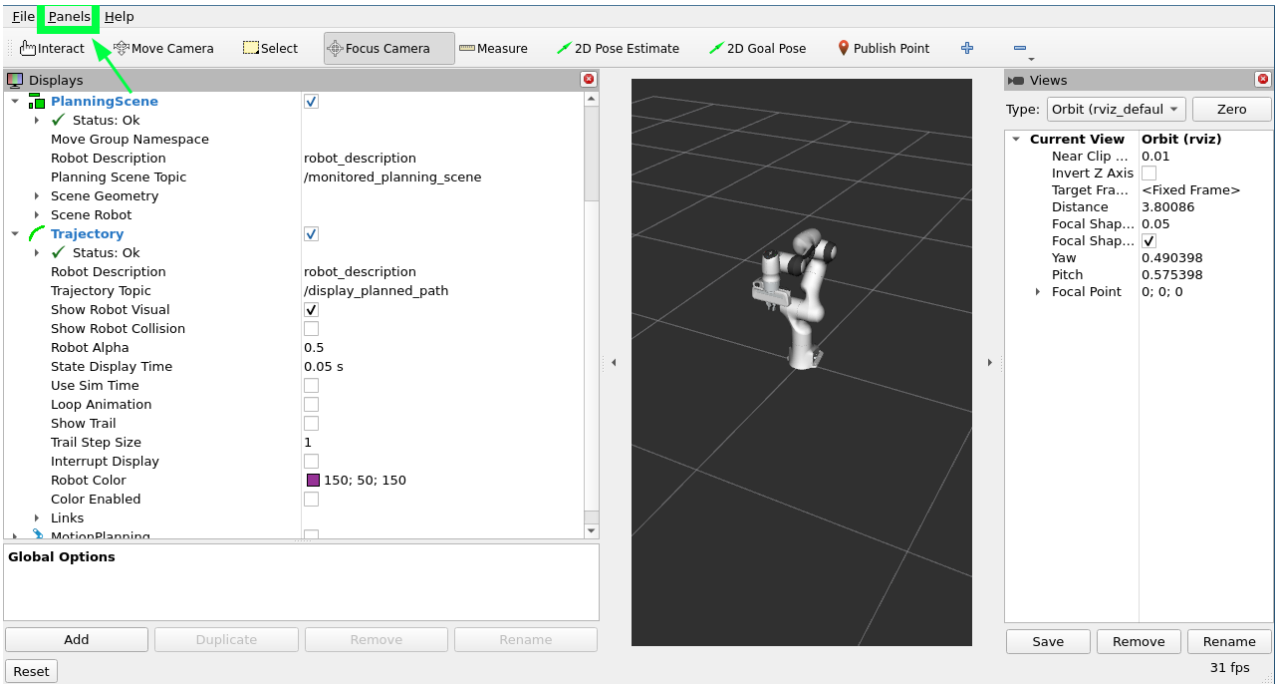
```
cd ~/ws_moveit2
source install/setup.bash
ros2 launch moveit2_tutorials demo.launch.py
```

Uncheck “MotionPlanning” in the “Displays” tab to hide it. We aren’t going to be using the “MotionPlanning” plugin for this next part.

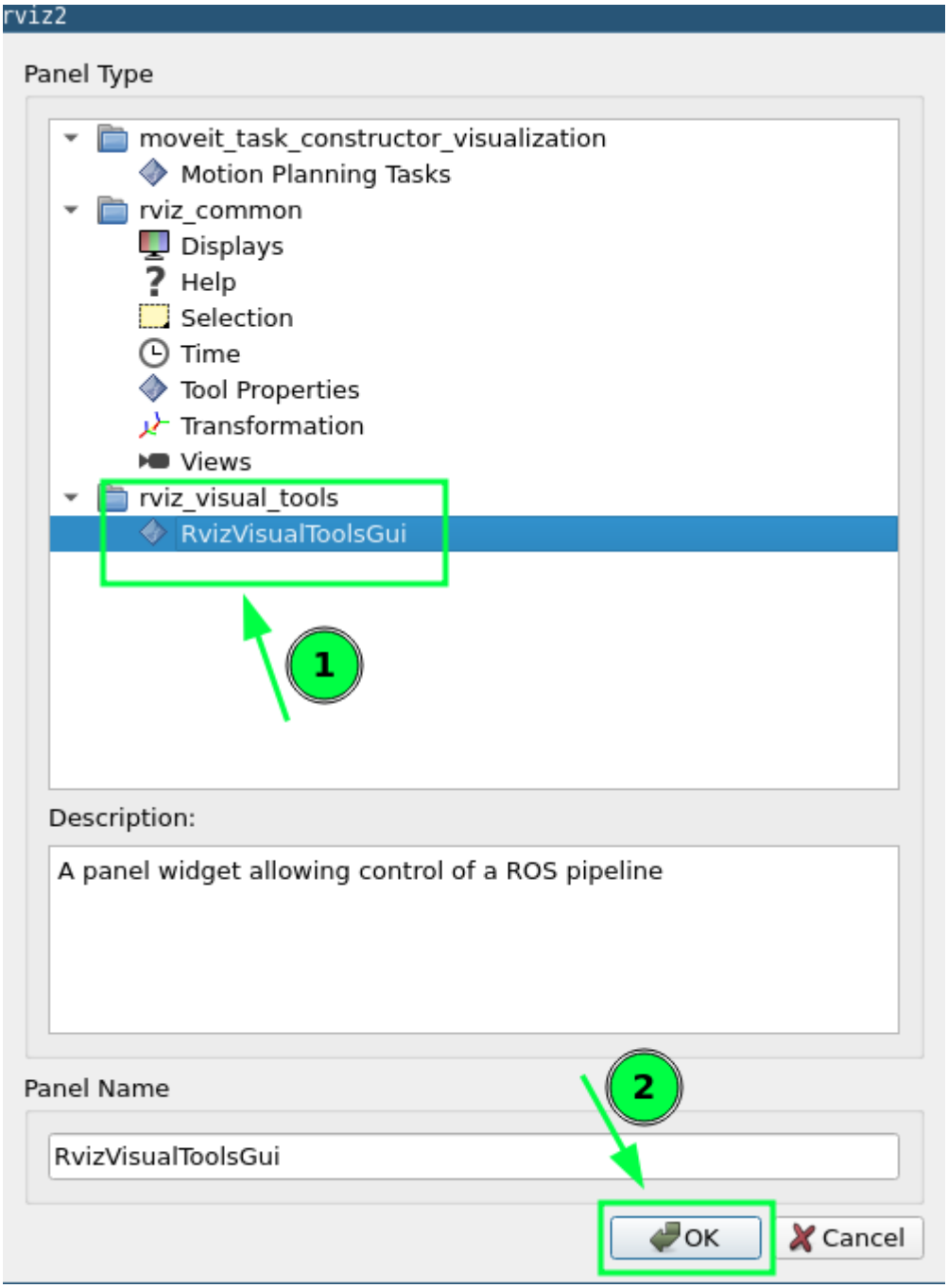


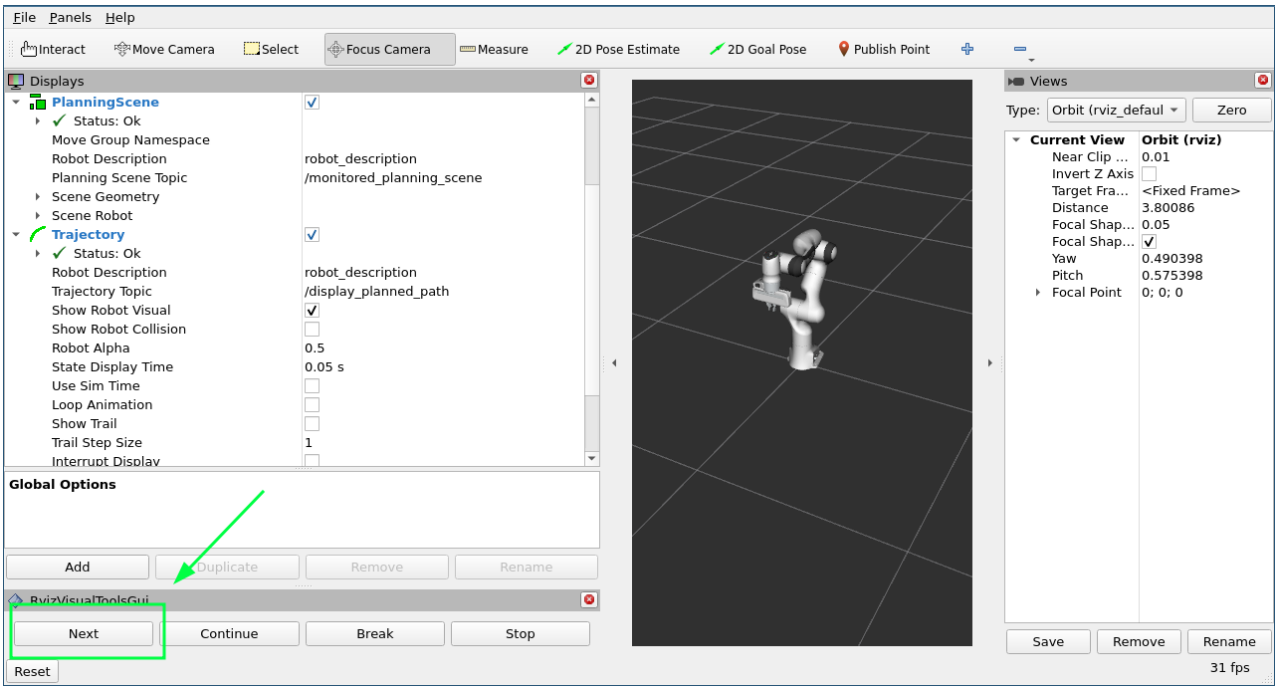


To add the buttons to interact with the prompts we added to our program open the dialog with the “Panels/Add New Panel” menu:

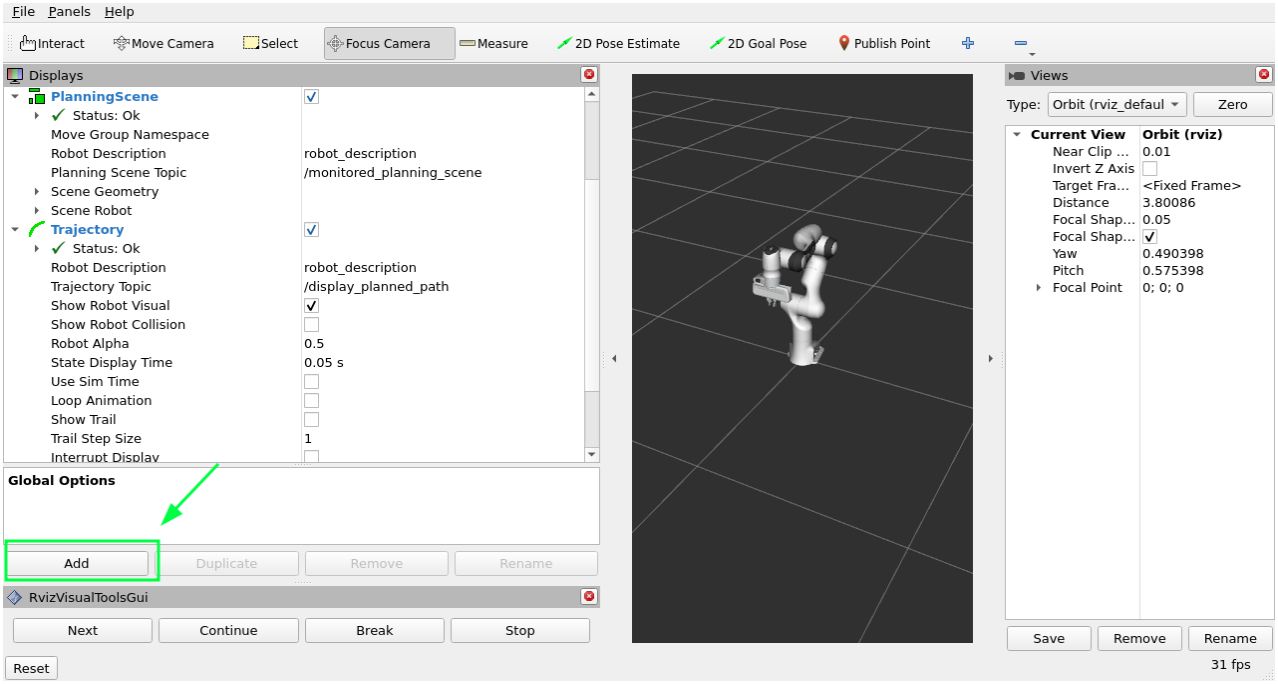


Then select `RvizVisualToolsGui` and click OK. This will create a new panel on the bottom left with a `Next` button we'll use later.

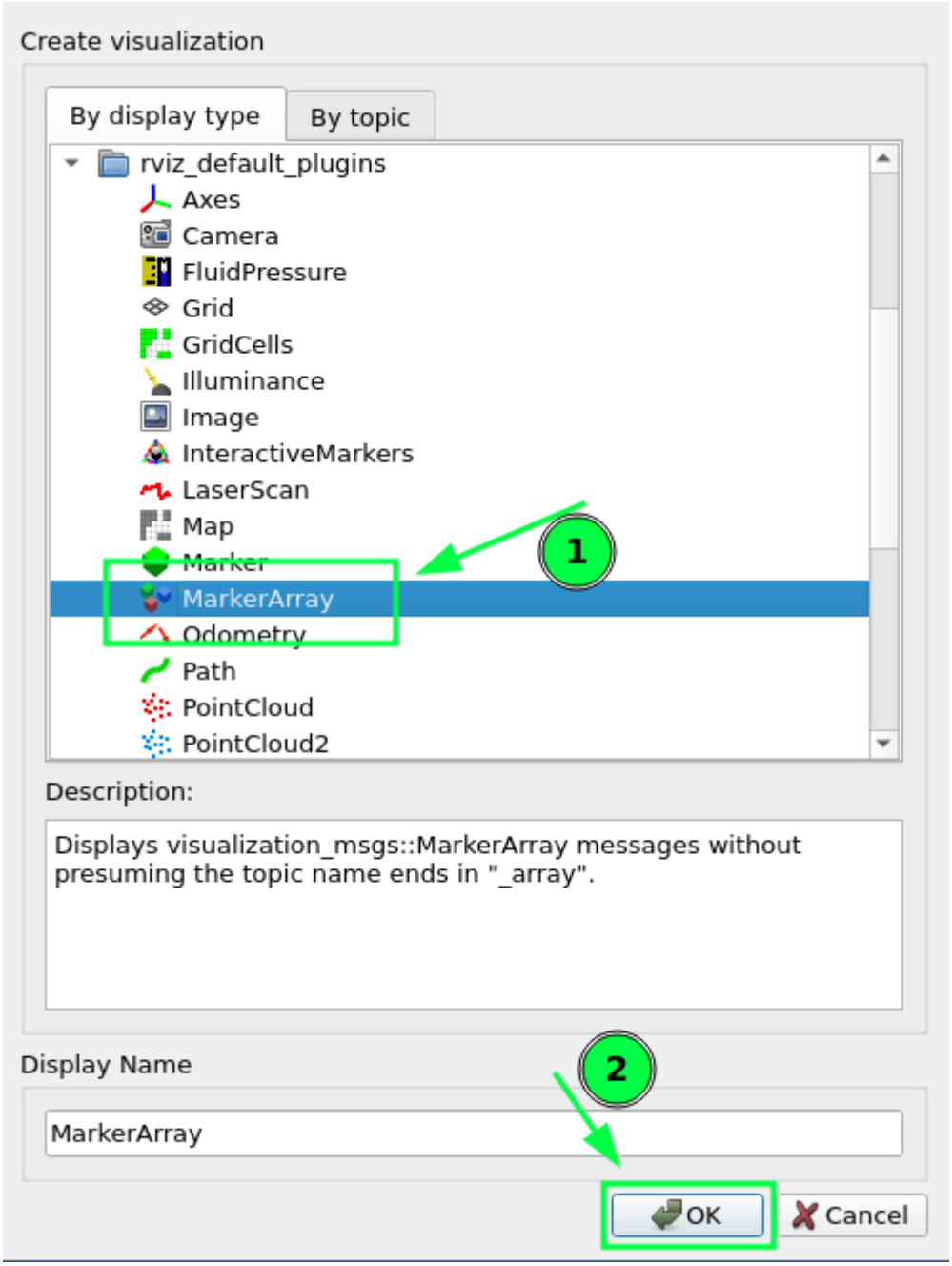

rviz2
Panel Type
moveit_task_constructor_visualization
Motion Planning Tasks
rviz_common
Displays
Help
Selection
Time
Tool Properties
Transformation
Views
rviz_visual_tools
RvizVisualToolsGui
1
Description:
A panel widget allowing control of a ROS pipeline
Panel Name
2
RvizVisualToolsGui
OK
Cancel



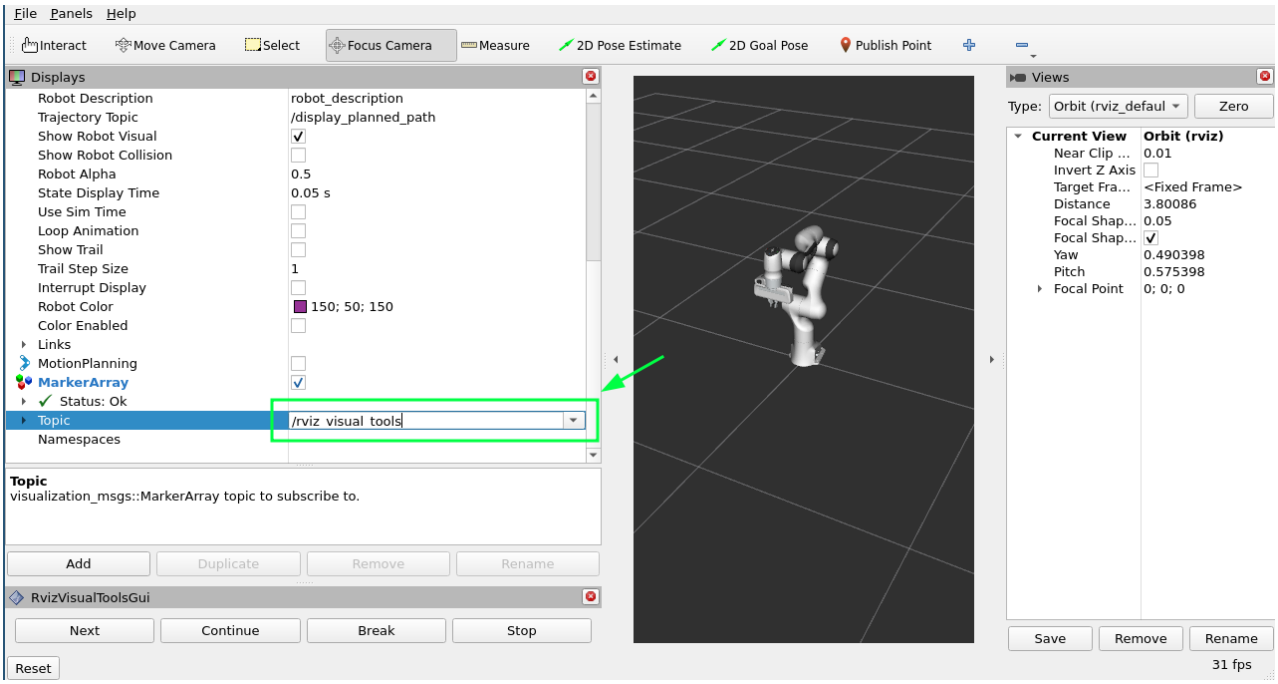
Finally, we need to add a `Marker Array` to render the visualizations we've added. Click on the "Add" Button in the "Displays" panel.



Select `Marker Array` and click `OK`.



Scroll to the bottom of the items in the Displays panel and edit the topic that the new Marker Array is using to `/rviz_visual_tools`.



You are now ready to run your new program with visualizations.

7 Run the Program_

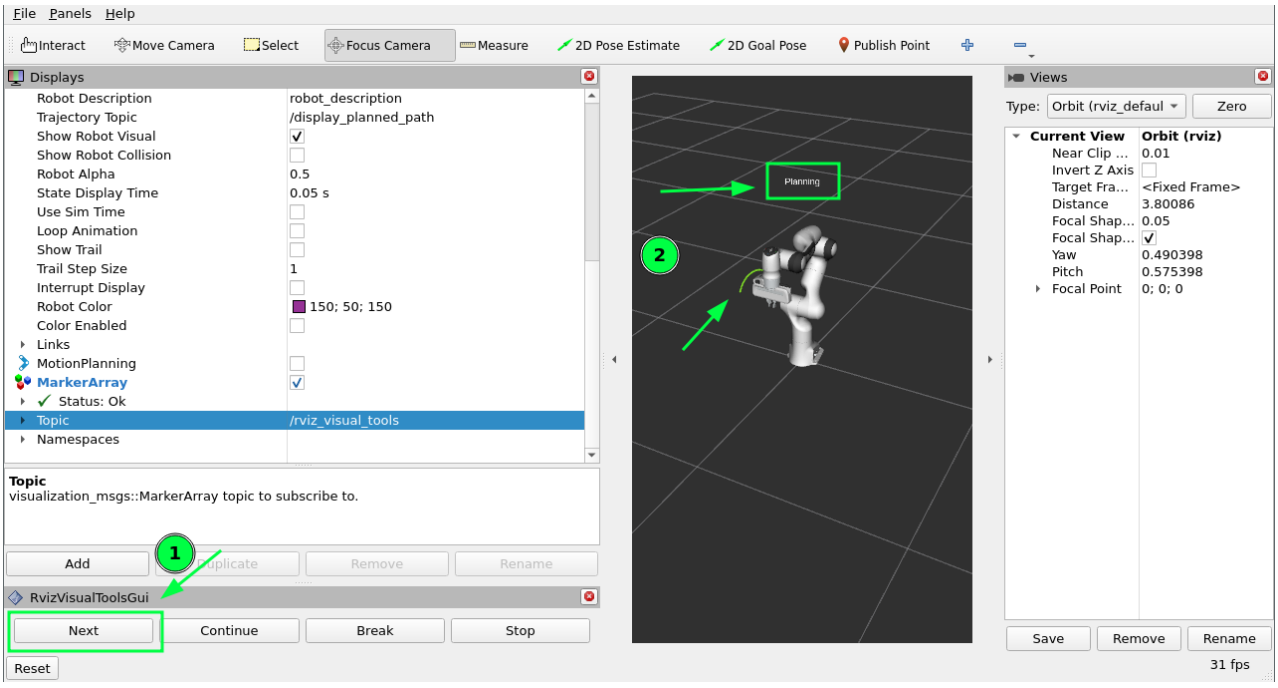
In a new terminal, go to the workspace, source the workspace, and run `hello_moveit`:

```
cd ~/ws_moveit2
source install/setup.bash
ros2 run hello_moveit hello_moveit
```

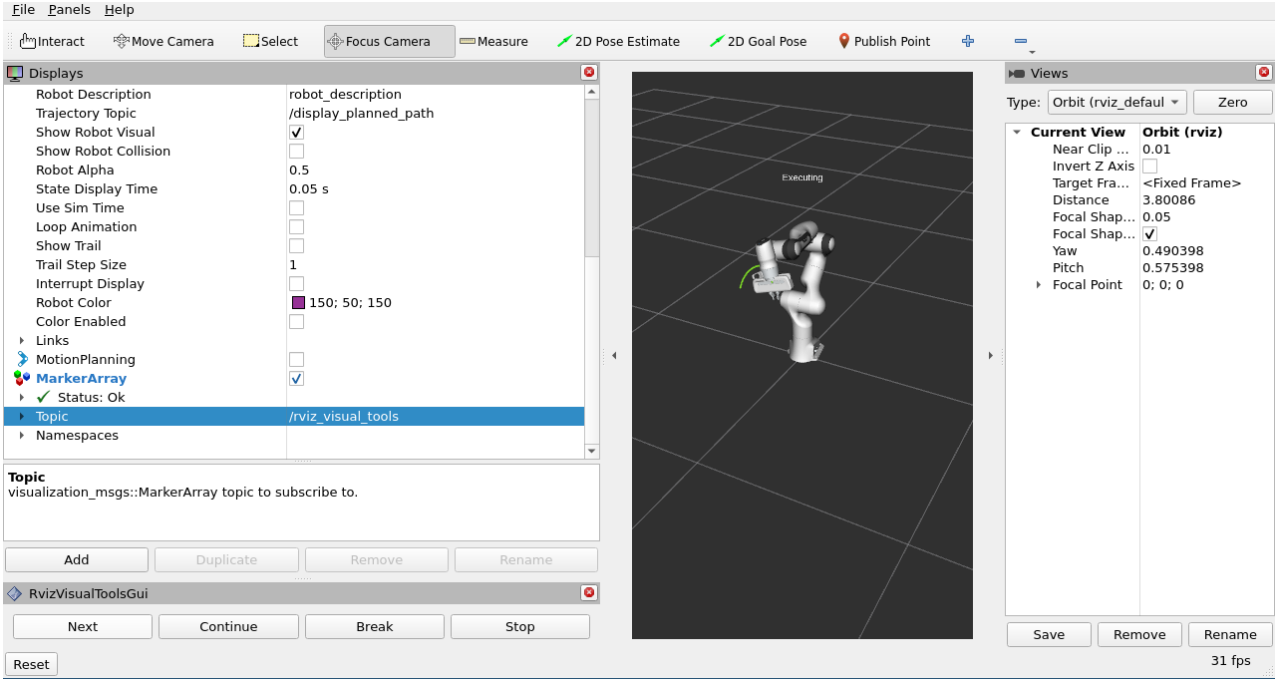
You'll notice that your program has stopped with a log that looks like this:

```
[INFO] [1652822889.492940200] [hello_moveit.remote_control]: Waiting to continue: Press
'Next' in the RvizVisualToolsGui window to plan
```

Click the `Next` button in RViz and see your application advance.



You'll see after you clicked the next button, your application planned, added a title above the robot, and drew a line representing the tool path. To continue, press `Next` again to see your robot execute the plan.



Summary_

You extended the program you wrote with MoveIt to interact with the Gui in RViz, allowing you to step through your program with a button, render some text above the robot, and display the tool path that you planned.

Further Reading_

- MoveItVisualTools has many more useful features for visualizing robot motions. You can read more about it here.
- There are also more examples of using `MoveItVisualTools` in MoveItCpp Tutorial.
- Here is a copy of the full hello_moveit.cpp source.

Next Step_

In the next tutorial Planning Around Objects, you will expand on the program you built here to add to the collision environment and see the robot plan with these changes.