23、ROS2 TF2坐标变换

1、TF2简介

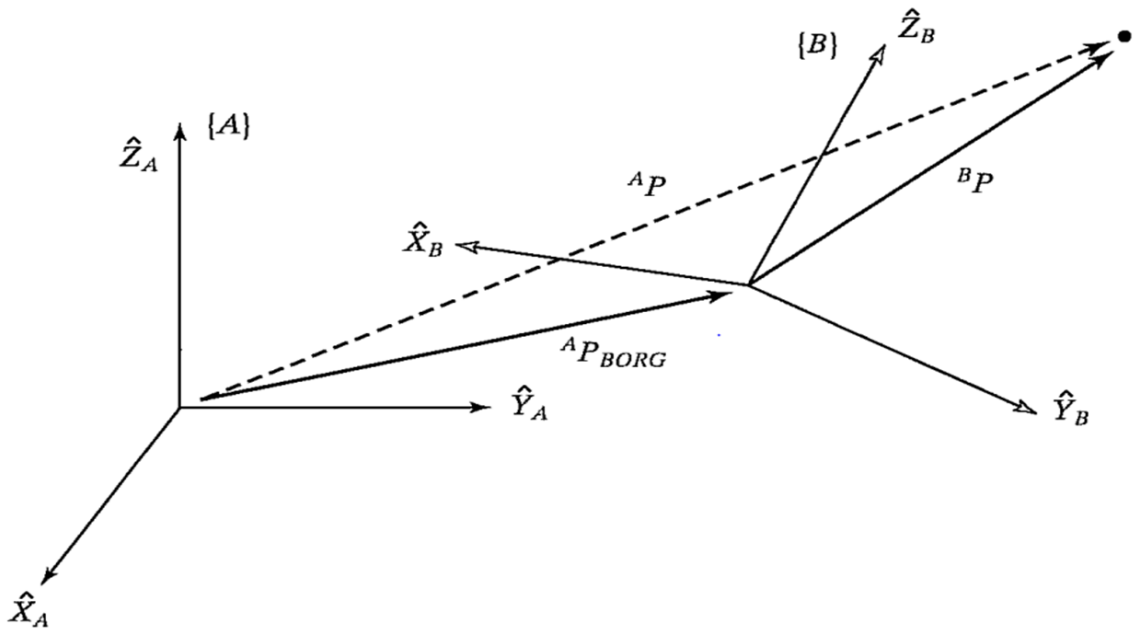
坐标系是我们非常熟悉的一个概念，也是机器人学中的重要基础，在一个完整的机器人系统中，会存在很多坐标系，这些坐标系之间的位置关系该如何管理？ROS给我们提供了一个坐标系的管理神器：TF2

TF系统参考文献：tf: The transform library | IEEE Conference Publication | IEEE Xplore

2、机器人中的坐标系

在移动机器人系统中，坐标系一样至关重要，比如一个移动机器人的中心点是基坐标系Base Link，雷达所在的位置叫做雷达坐标系laser link，机器人要移动，里程计会累积位置，这个位置的参考系叫做里程计坐标系odom，里程计又会有累积误差和漂移，绝对位置的参考系叫做地图坐标系map。

一层一层坐标系之间关系复杂，有一些是相对固定的，也有一些是不断变化的，看似简单的坐标系也在空间范围内变得复杂，良好的坐标系管理系统就显得格外重要。



关于坐标系变换关系的基本理论，在每一本机器人学的教材中都会有讲解，可以分解为**平移和旋转**两个部分，通过一个四乘四的矩阵进行描述，在空间中画出坐标系，那两者之间的变换关系，其实就是向量的数学描述。

ROS中TF功能的底层原理，就是对这些数学变换进行了封装，详细的理论知识大家可以参考机器人学的教材，我们主要讲解TF坐标管理系统的使用方法。

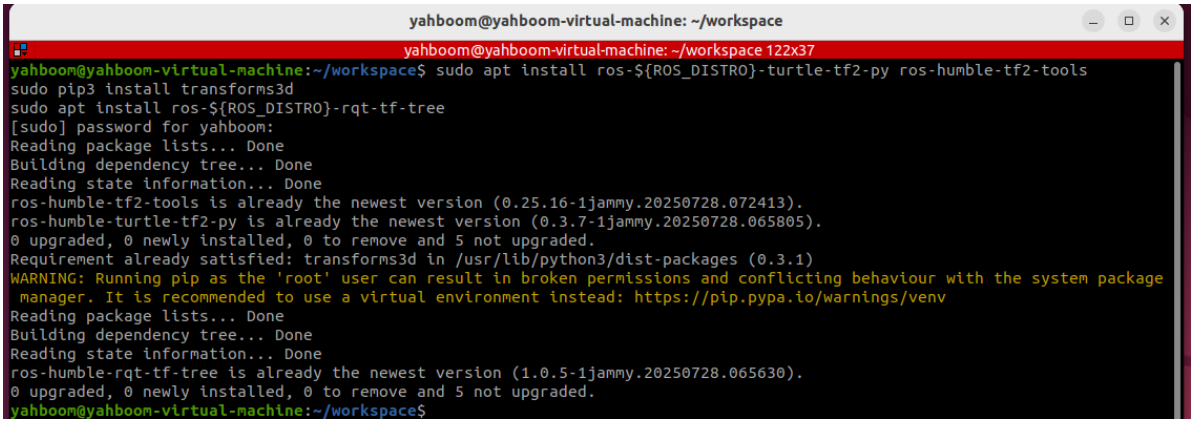
3、TF命令行操作

我们先通过两只小海龟的示例，了解下基于坐标系的一种机器人跟随算法。**为方便演示，本节课程最好选择在虚拟机中操作**

3.1、安装相关工具

这个示例需要我们先安装相应的功能包、tf海龟模拟器案例、tf树可视化工具

```
sudo apt install ros-${ROS_DISTRO}-turtle-tf2-py ros-humble-tf2-tools
sudo pip3 install transforms3d
sudo apt install ros-${ROS_DISTRO}-rqt-tf-tree
```

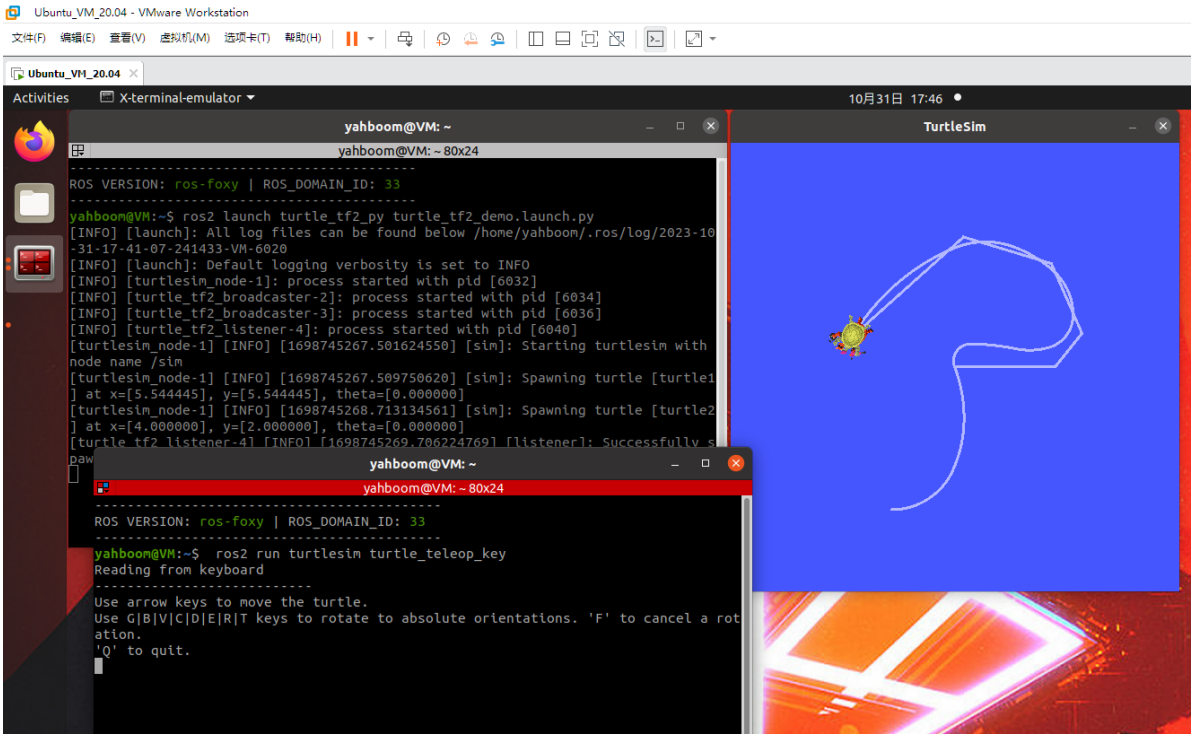


3.2、启动

然后就可以通过一个launch文件启动，之后我们可以控制其中的一只小海龟，另外一只小海龟会自动跟随运动。

```
ros2 launch turtle_tf2_py turtle_tf2_demo.launch.py
ros2 run turtlesim turtle_teleop_key
```

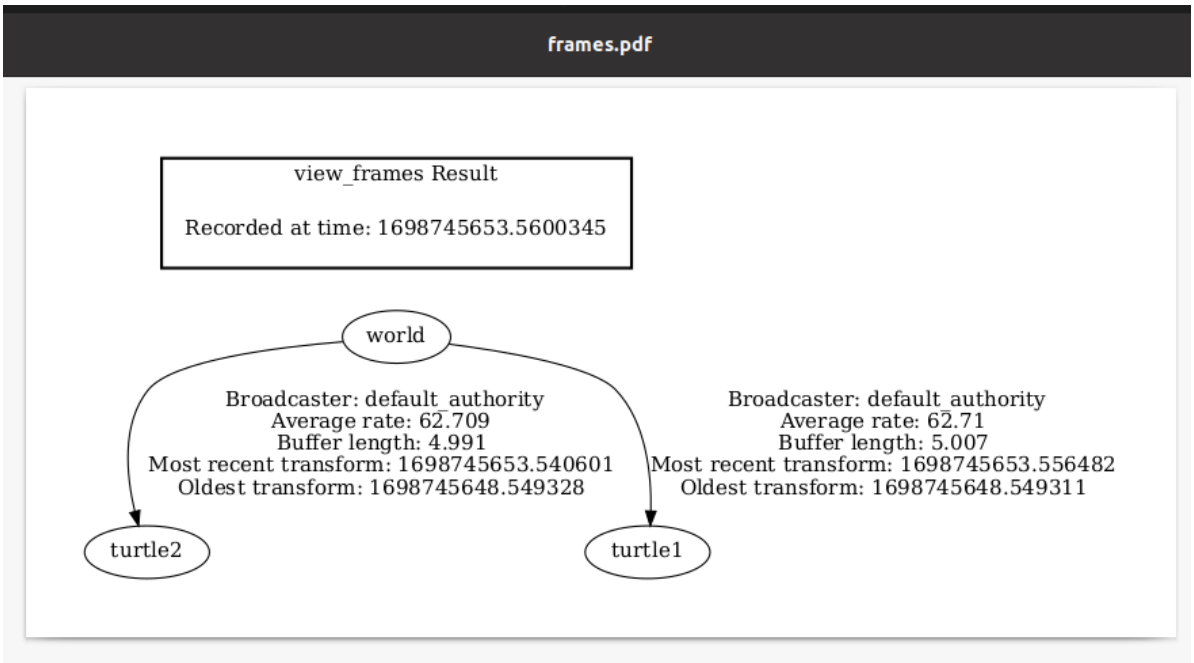
当我们控制一只海龟运动时，另外一只海龟也会跟随运动。



3.3、查看TF树

```
ros2 run rqt_tf_tree rqt_tf_tree
```

可以在rqt窗口中看到TF变换树

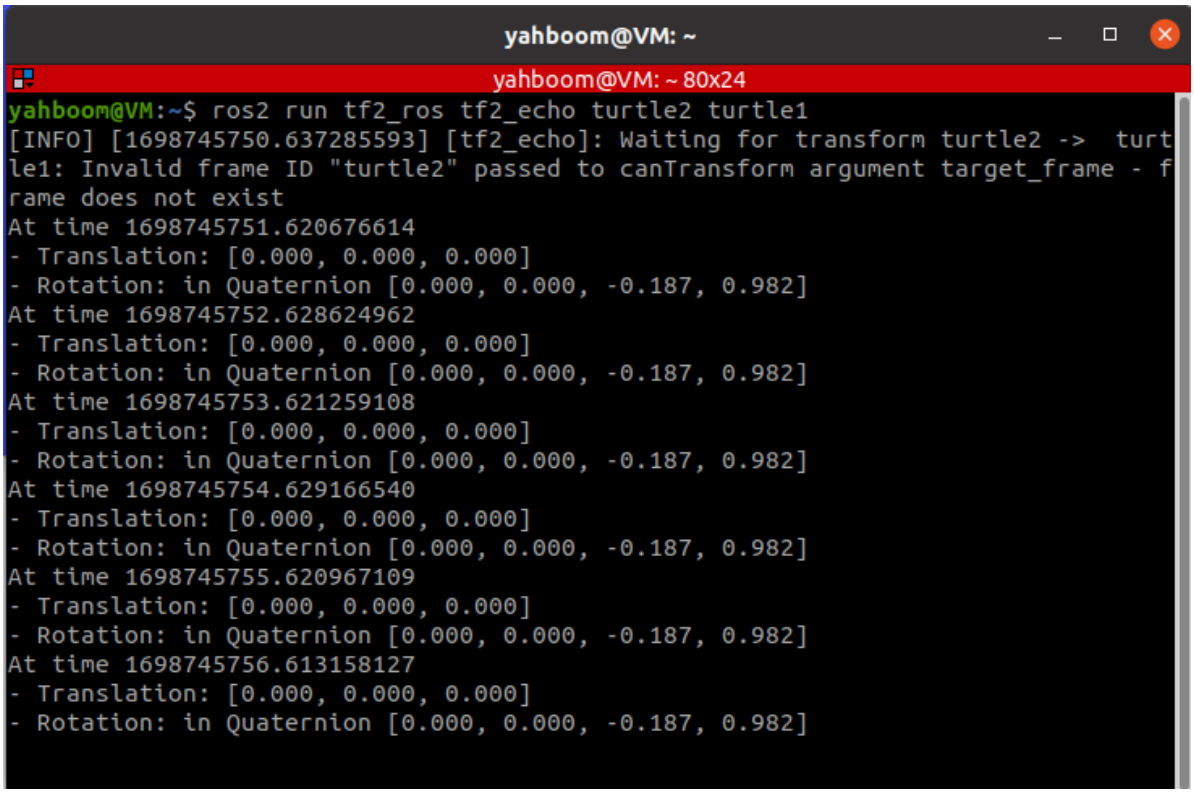


3.4、查询坐标变换信息

只看到坐标系的结构还不行，如果我们想要知道某两个坐标系之间的具体关系，可以通过tf2_echo这个工具查看：

```
ros2 run tf2_ros tf2_echo turtle2 turtle1
```

运行成功后，终端中就会循环打印坐标系的变换数值了

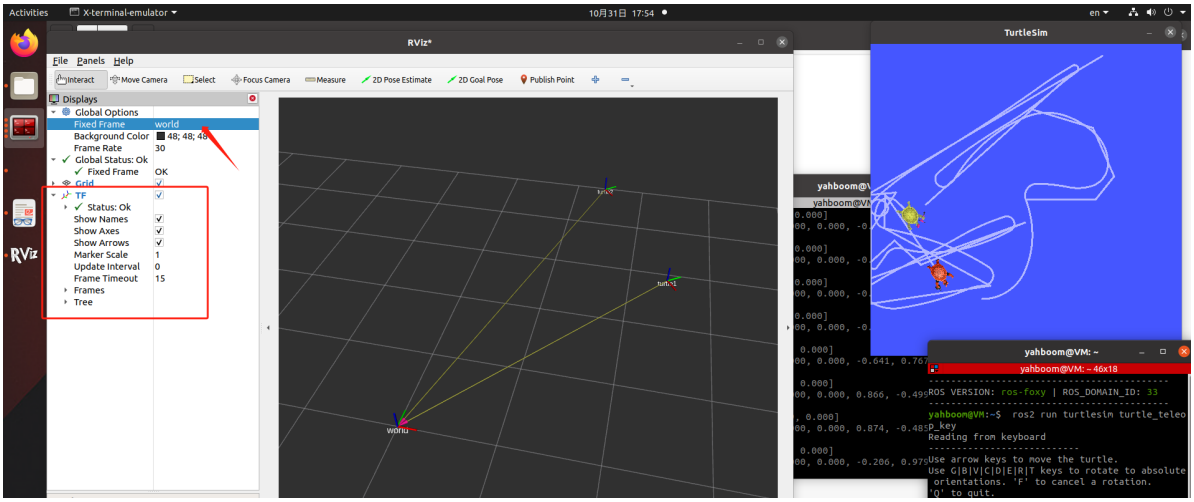


3.5、坐标系可视化

- 运行rviz2，然后添加TF显示插件

```
rviz2
```

rivz2中设置参考坐标系为：world，添加TF显示，再让小海龟动起来，Rviz中的坐标轴就会开始运动，这样是不是更加直观了呢！



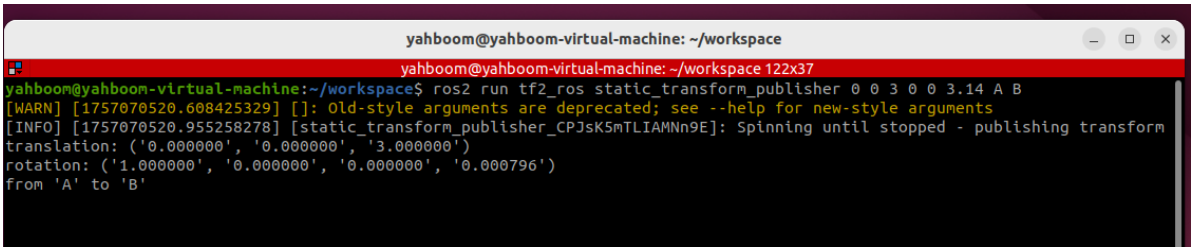
4、静态坐标变换

所谓静态坐标变换，是指两个坐标系之间的相对位置是固定的。如雷达和base_link之间的位置是固定的。

示例：**为方便演示，本节课程最好选择在虚拟机中操作**

4.1、发布A到B的位姿

```
ros2 run tf2_ros static_transform_publisher 0 0 3 0 0 3.14 A B
```



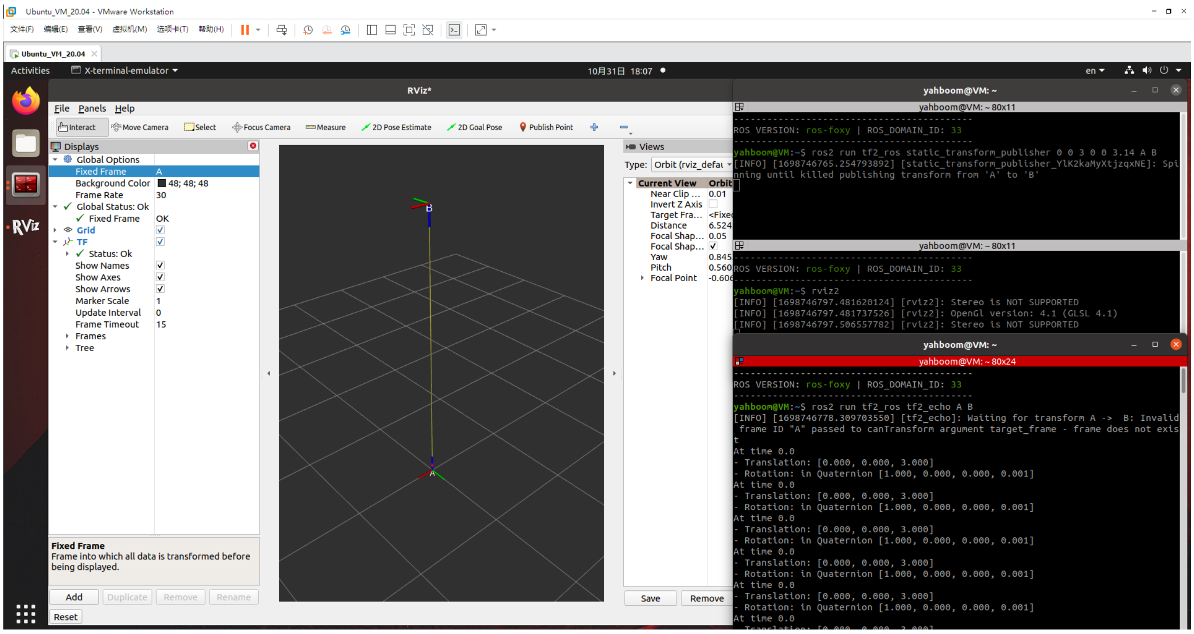
4.2、监听/获取TF关系

```
ros2 run tf2_ros tf2_echo A B
```

4.3、rivz可视化

- 运行rviz2，然后添加TF显示插件

```
rviz2
```



5、案例介绍

上节课程中讲解了系统提供的小海龟跟随案例中的TF关系，这节课我们自己实现该功能。

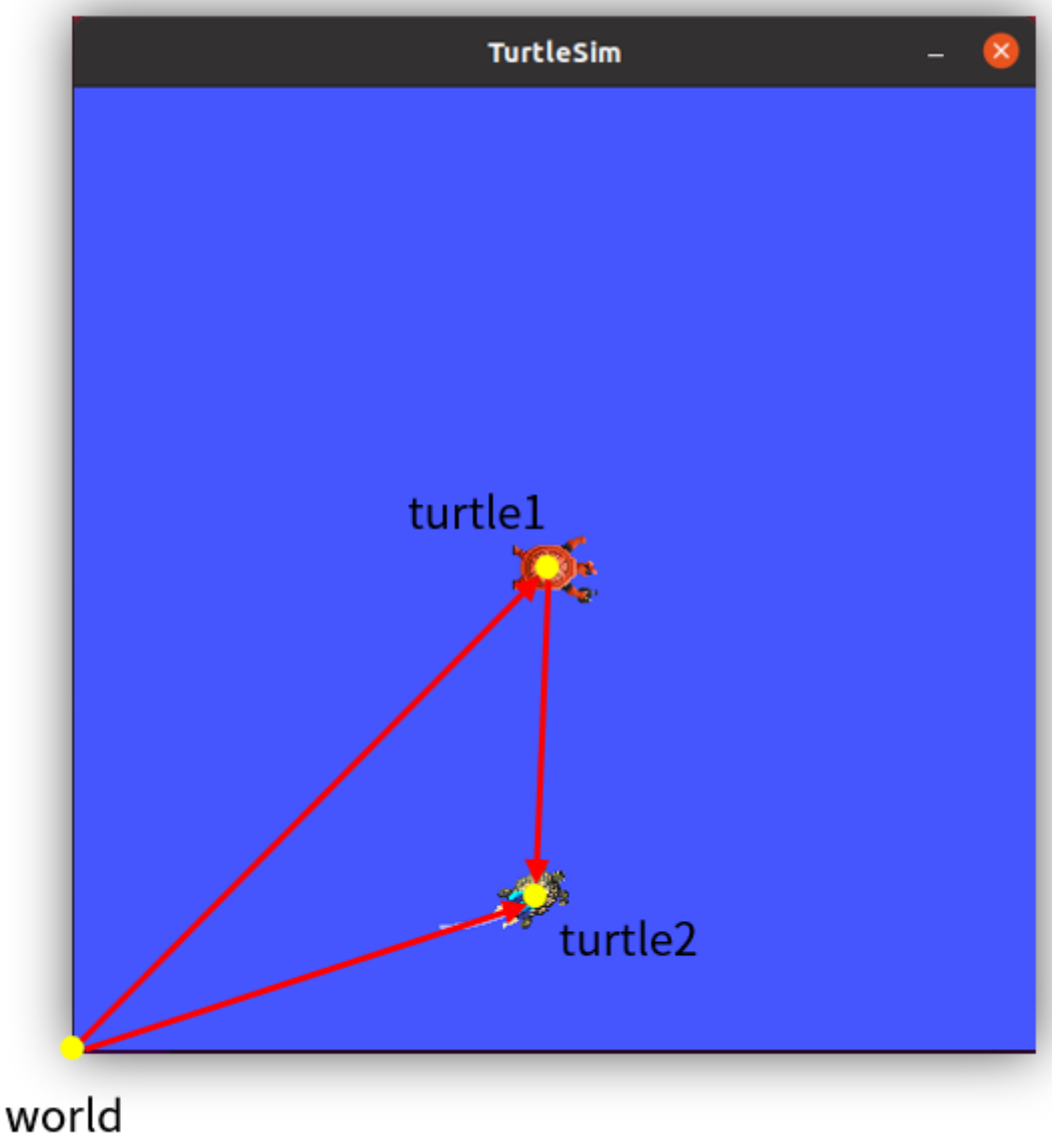
课程内容：

- 通过编程的方式实现小海龟跟随案例
- 掌握通过编程实现动态广播器
- 掌握通过编程实现监听坐标系之间的坐标变换
- 掌握通过PID控制将物理量（距离、角度）转换成速度控制量

进阶内容：

- 理解TF系统跨时间维度变换坐标系功能

6、海龟跟随案例实现原理分析



在两只海龟的仿真器中，我们可以定义三个坐标系，比如仿真器的全局参考系叫做world，turtle1和turtle2坐标系在两只海龟的中心点，这样，turtle1和world坐标系的相对位置，就可以表示海龟1的位置，海龟2也同理。

要实现海龟2向海龟1运动，我们在两者中间做一个连线，再加一个箭头，怎么样，是不是有想起高中时学习的向量计算？我们说坐标变换的描述方法就是向量，所以在这个跟随例程中，用TF就可以很好的解决。

向量的长度表示距离，方向表示角度，有了距离和角度，我们随便设置一个时间，不就可以计算得到速度了么，然后就是速度话题的封装和发布，海龟2也就可以动起来了。

所以这个例程的核心就是通过坐标系实现向量的计算，两只海龟还会不断运动，这个向量也得按照某一个周期计算，这就得用上TF的动态广播与监听了。

7、新建功能包

1. 在工作空间的src目录下新建功能包用于存放我们的文件

```
ros2 pkg create pkg_tf --build-type ament_python --dependencies rclpy --node-
name turtle_tf_broadcaster
```

执行完上述命令，会创建pkg_tf功能包，同时会创建一个turtle_tf_broadcaster的节点，并且已经配置好相关的配置文件，在【turtle_tf_broadcaster.py】文件中添加如下代码：

```
import rclpy                                            # ROS2 Python接口库
from rclpy.node import Node                             # ROS2 节点类
from geometry_msgs.msg import TransformStamped          # 坐标变换消息
import tf_transformations                               # TF坐标变换库
from tf2_ros import TransformBroadcaster                # TF坐标变换广播器
from turtlesim.msg import Pose                          # turtlesim小海龟位置消息

class TurtleTFBroadcaster(Node):

    def __init__(self, name):
        super().__init__(name)                                  # ROS2节点父类初始化

        self.declare_parameter('turtlename', 'turtle')          # 创建一个海龟名称的参数
        self.turtlename = self.get_parameter(                   # 优先使用外部设置的参数值，否则用默认值
            'turtlename').get_parameter_value().string_value

        self.tf_broadcaster = TransformBroadcaster(self)        # 创建一个TF坐标变换的广播对象并初始化

        self.subscription = self.create_subscription(           # 创建一个订阅者，订阅海龟的位置消息
            Pose,
            f'/{self.turtlename}/pose',                         # 使用参数中获取到的海龟名称
            self.turtle_pose_callback, 1)

    def turtle_pose_callback(self, msg):                                # 创建一个处理海龟位置消息的回调函数，将位置消息转变成坐标变换
        transform = TransformStamped()                                  # 创建一个坐标变换的消息对象

        transform.header.stamp = self.get_clock().now().to_msg()        # 设置坐标变换消息的时间戳
        transform.header.frame_id = 'world'                             # 设置一个坐标变换的源坐标系
        transform.child_frame_id = self.turtlename                      # 设置一个坐标变换的目标坐标系
        transform.transform.translation.x = msg.x                       # 设置坐标变换中的X、Y、Z向的平移
        transform.transform.translation.y = msg.y
        transform.transform.translation.z = 0.0
        q = tf_transformations.quaternion_from_euler(0, 0, msg.theta) # 将欧拉角转换为四元数（roll, pitch, yaw）
        transform.transform.rotation.x = q[0]                           # 设置坐标变换中的X、Y、Z向的旋转（四元数）
        transform.transform.rotation.y = q[1]
        transform.transform.rotation.z = q[2]
        transform.transform.rotation.w = q[3]

        # Send the transformation
```

```
        self.tf_broadcaster.sendTransform(transform)      # 广播坐标变换，海龟位置变化后，将及时更新坐标变换信息

def main(args=None):
    rclpy.init(args=args)                                 # ROS2 Python接口初始化
    node = TurtleTFBroadcaster("turtle_tf_broadcaster")   # 创建ROS2节点对象并进行初始化
    rclpy.spin(node)                                      # 循环等待ROS2退出
    node.destroy_node()                                   # 销毁节点对象
    rclpy.shutdown()                                      # 关闭ROS2 Python接口
```

2、接下来在turtle_tf_broadcaster.py同级目录下新建【turtle_following.py】文件，添加如下代码：

```
import math
import rclpy                                              # ROS2 Python接口库
from rclpy.node import Node                               # ROS2 节点类
import tf_transformations                                 # TF坐标变换库
from tf2_ros import TransformException                    # TF左边变换的异常类
from tf2_ros.buffer import Buffer                         # 存储坐标变换信息的缓冲类
from tf2_ros.transform_listener import TransformListener  # 监听坐标变换的监听器类
from geometry_msgs.msg import Twist                       # ROS2 速度控制消息
from turtlesim.srv import Spawn                           # 海龟生成的服务接口
class TurtleFollowing(Node):

    def __init__(self, name):
        super().__init__(name)                                      # ROS2节点父类初始化

        self.declare_parameter('source_frame', 'turtle1')           # 创建一个源坐标系名的参数
        self.source_frame = self.get_parameter(                     # 优先使用外部设置的参数值，否则用默认值
            'source_frame').get_parameter_value().string_value

        self.tf_buffer = Buffer()                                   # 创建保存坐标变换信息的缓冲区
        self.tf_listener = TransformListener(self.tf_buffer, self)  # 创建坐标变换的监听器

        self.spawner = self.create_client(Spawn, 'spawn')           # 创建一个请求产生海龟的客户端
        self.turtle_spawning_service_ready = False                  # 是否已经请求海龟生成服务的标志位
        self.turtle_spawned = False                                 # 海龟是否产生成功的标志位

        self.publisher = self.create_publisher(Twist, 'turtle2/cmd_vel', 1) # 创建跟随运动海龟的速度话题

        self.timer = self.create_timer(1.0, self.on_timer)         # 创建一个固定周期的定时器，控制跟随海龟的运动

    def on_timer(self):
        from_frame_rel = self.source_frame                          # 源坐标系
        to_frame_rel   = 'turtle2'                                  # 目标坐标系
```

```
        if self.turtle_spawning_service_ready:                          # 如果已经请求海
龟生成服务
            if self.turtle_spawned:                                     # 如果跟随海龟已
经生成
                try:
                    now = rclpy.time.Time()                             # 获取ROS系统的
当前时间
                    trans = self.tf_buffer.lookup_transform(            # 监听当前时刻源
坐标系到目标坐标系的坐标变换
                        to_frame_rel,
                        from_frame_rel,
                        now)
                except TransformException as ex:                        # 如果坐标变换获
取失败，进入异常报告
                    self.get_logger().info(
                        f'Could not transform {to_frame_rel} to
{from_frame_rel}: {ex}')
                    return

                msg = Twist()                                           # 创建速度控制消
息
                scale_rotation_rate = 1.0                               # 根据海龟角度，
计算角速度
                msg.angular.z = scale_rotation_rate * math.atan2(
                    trans.transform.translation.y,
                    trans.transform.translation.x)

                scale_forward_speed = 0.5                               # 根据海龟距离，
计算线速度
                msg.linear.x = scale_forward_speed * math.sqrt(
                    trans.transform.translation.x ** 2 +
                    trans.transform.translation.y ** 2)

                self.publisher.publish(msg)                             # 发布速度指令，
海龟跟随运动
            else:                                                       # 如果跟随海龟没
有生成
                if self.result.done():                                  # 查看海龟是否生
成
                    self.get_logger().info(
                        f'Successfully spawned {self.result.result().name}')
                    self.turtle_spawned = True
                else:                                                   # 依然没有生成跟
随海龟
                    self.get_logger().info('Spawn is not finished')
        else:                                                           # 如果没有请求海
龟生成服务
            if self.spawner.service_is_ready():                         # 如果海龟生成服
务器已经准备就绪
                request = Spawn.Request()                               # 创建一个请求的
数据
                request.name = 'turtle2'                                # 设置请求数据的
内容，包括海龟名、xy位置、姿态
                request.x = float(4)
                request.y = float(2)
                request.theta = float(0)
```

```
                self.result = self.spawner.call_async(request)      # 发送服务请求
                self.turtle_spawning_service_ready = True           # 设置标志位，表示已经发送请求
            else:
                self.get_logger().info('Service is not ready')      # 海龟生成服务器还没准备就绪的提示


def main(args=None):
    rclpy.init(args=args)                           # ROS2 Python接口初始化
    node = TurtleFollowing("turtle_following")  # 创建ROS2节点对象并进行初始化
    rclpy.spin(node)                                # 循环等待ROS2退出
    node.destroy_node()                             # 销毁节点对象
    rclpy.shutdown()                                # 关闭ROS2 Python接口
```

3、在pkg_tf功能包下新建launch文件夹，在launch文件夹内新建【turtle_following.launch.py】文件，添加如下内容：

```
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node

def generate_launch_description():

    return LaunchDescription([
        DeclareLaunchArgument('source_frame', default_value='turtle1', description='Target frame name.'),
        Node(
            package='turtlesim',
            executable='turtlesim_node',
        ),
        Node(
            package='pkg_tf',
            executable='turtle_tf_broadcaster',
            name='broadcaster1',
            parameters=[
                {'turtlename': 'turtle1'}
            ]
        ),
        Node(
            package='pkg_tf',
            executable='turtle_tf_broadcaster',
            name='broadcaster2',
            parameters=[
                {'turtlename': 'turtle2'}
            ]
        ),
        Node(
            package='pkg_tf',
            executable='turtle_following',
            name='listener',
            parameters=[
                {'source_frame': LaunchConfiguration('source_frame')}
            ]
```

```
        ),
    ])
```

8、编辑配置文件

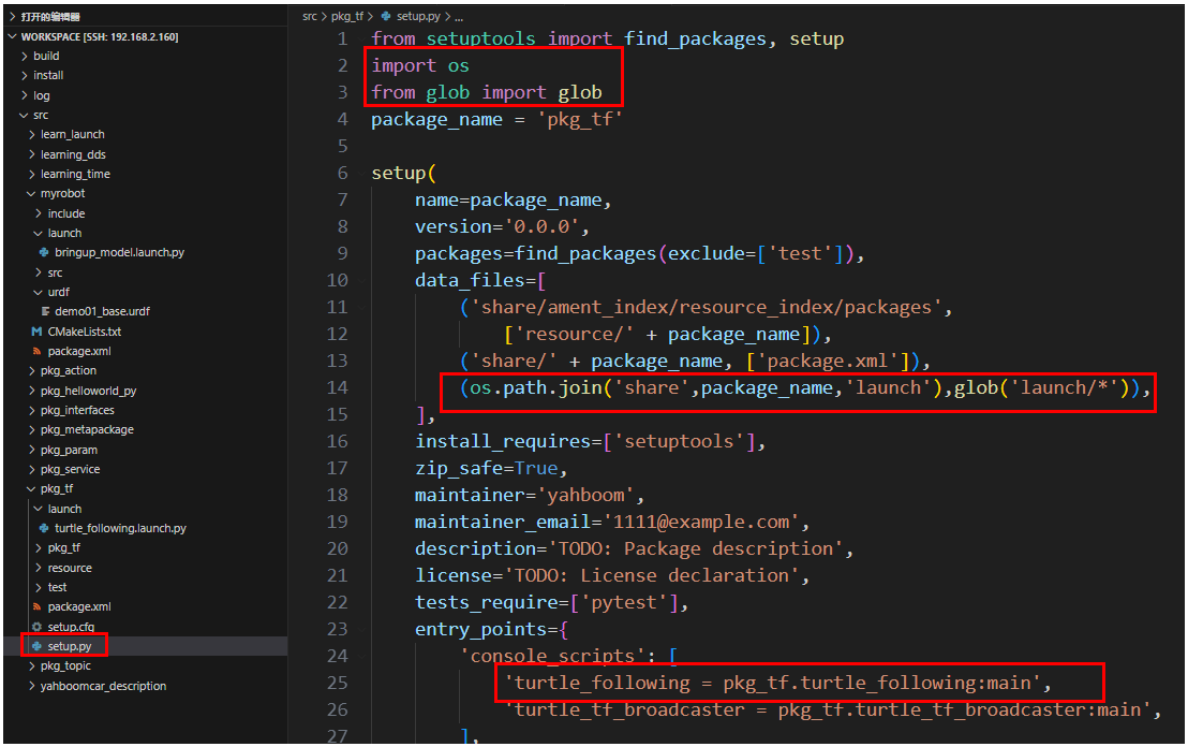
8.1、setup.py中配置

- 导入相关库

```
import os
from glob import glob
```

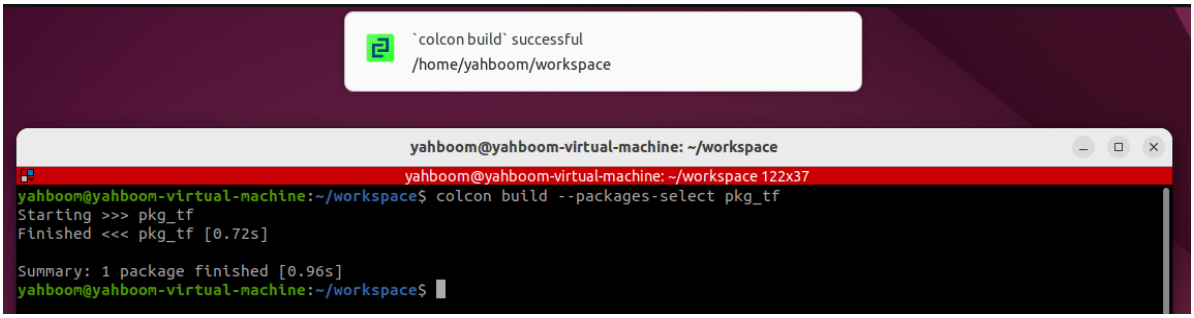
- 添加turtle_following节点信息，并添加一下命令将launch文件拷贝到install中共享目录中

```
(os.path.join('share',package_name,'launch'),glob('launch/*')),
```



9、编译功能包

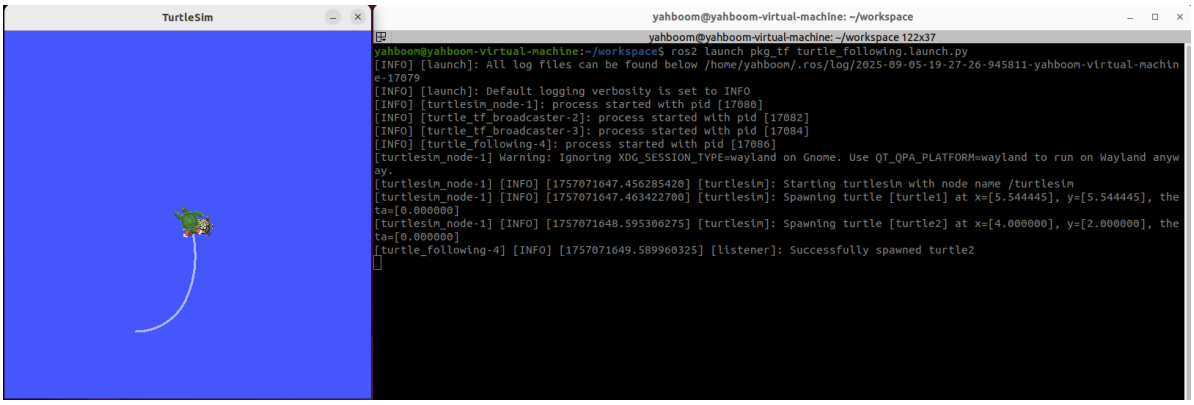
```
colcon build --packages-select pkg_tf
```



10、运行程序

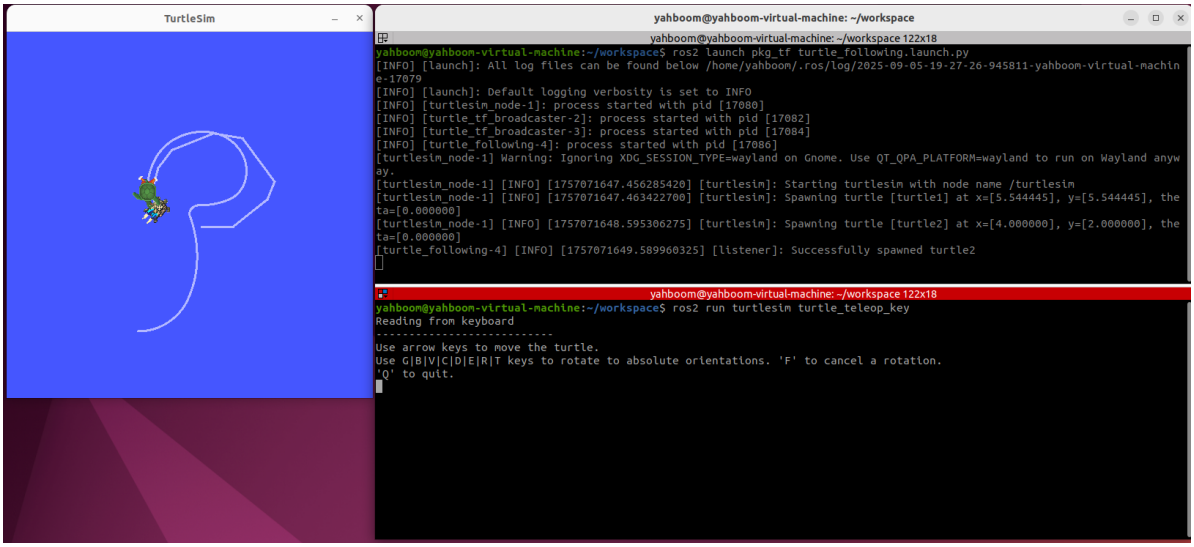
- 刷新终端环境变量，然后运行

```
ros2 launch pkg_tf turtle_following.launch.py
```



- 启动海龟键盘控制节点，控制第一只小海龟运动，第二只海龟会自动跟随

```
ros2 run turtlesim turtle_teleop_key
```



在此终端内按键盘的上下左右键可以控制其中的一个小乌龟运动，然后另外一个小乌龟会跟着运动直到它们重合。

11、进阶内容

- 理解TF的跨时间维度变换能力

  Buffer能够通过缓冲区自动缓存过去10s内TF系统内所有的TF变换关系（可以通过Buffer构造参数自己设置任意的缓存时长），缓冲区内所有的变换以时间戳为单位，所有变换都是连续可追溯的，即使两个坐标系处于不同的时间点，也能查找出之间的坐标变换关系，这一点可以参考设计TF系统的参考文献，有详细的原理讲解（文献在本节课程文件夹下、或者通过本节开头时的链接）

```
self.tf_buffer = Buffer()                                  # 创建保存坐标变换信息的缓冲区
self.tf_listener = TransformListener(self.tf_buffer, self)  # 创建坐标变换的监听器
```

- 以下是对我们上述海龟跟随案例的补充，红色箭头表示可以跨时间查找不同时间点的两个坐标系之间的变换

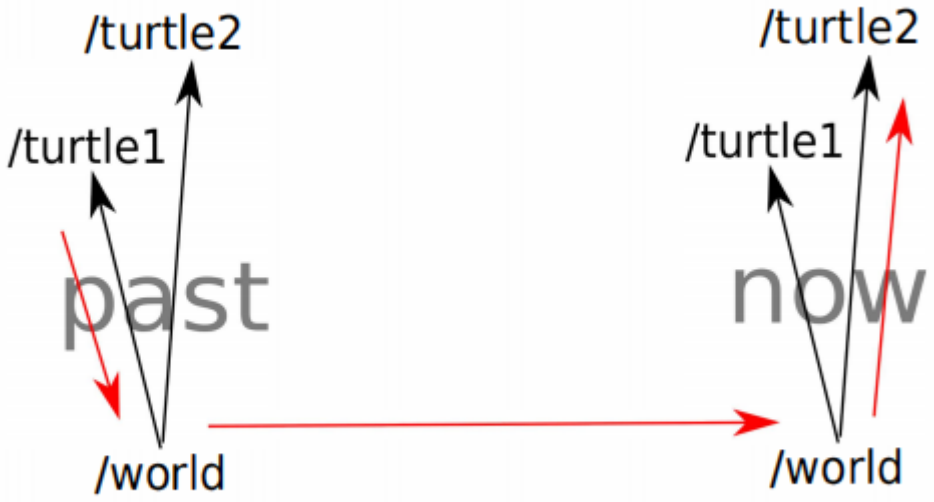


Fig. 4: A simple *tf* tree from a core ROS tutorial, with debugging information.

Long Term Data Storage: The above example assumes that you have the ability to transform between frames a and b at time 0 and b and c at time 1. This will only work if the length of the buffer in your *Listener* module is long enough to encompass both time 0 and time 1. To be able to store data for a long time, it should be transformed into the fixed frame, b, when saved. Then the source frame is the same as frame b and consequently the transform between the two is the identity, leaving only $T_{b@t_1}^{c@t_1}$ needing to be computed to find the current location of any previously observed object. This simple case is common in many robotic systems because they do not have the tools to compute the more complicated cases. This will only work when the assumptions about where the identity transform can be used is maintained.