DAY2
과제 2번 보고서

2023741024
로봇학부 박건후

목차

(1) qnode.hpp파일 설명

```
#include <rclcpp/rclcpp.hpp>
//3번 과제에서 인클루드 했던 대로 인클루드함
#include <geometry_msgs/msg/twist.hpp>
#include <turtlesim/srv/set_pen.hpp>
#include <std_srvs/srv/empty.hpp>
#endif
#include <QThread>

namespace Ui { class MainWindow; }
/*****************************************************************
** Class
*****************************************************************
class publisher: public rclcpp::Node{ //publisher 클래스 선언
    //3번 과제에서 선언한 포인터 그대로 사용
    rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr pub;
    rclcpp::Client<turtlesim::srv::SetPen>::SharedPtr set_pen;

    public:
    Ui:: MainWindow* ui; //ui에 접근하기 위해 포인터 선언하고 publisher 내부
    publisher();
    void publish_message(int flag); //버튼을 누르는 것으로 어떤 동작을 수행하
};
```

과제 3번에서 구현했던 기능에 더해 qt와 ros를 연동시키는 것이 핵심입니다. 그리고 qnode는 main_window에서 하는 qt작업과 ros를 연결시키는 다리 역할을 하는 느낌입니다. 그렇기에 qnode를 먼저 설명드려 ros가 돌아가는 내부적인 부분을 먼저 설명드리고 겉부분과 관련된 main_window에 대해 설명드리겠습니다.

큰 틀은 이렇습니다. main_window에서 QNode클래스의 qnode 포인터를 가지고 있고 QNode클래스를 통해 ros관련 구현부를 가리켜서 ros 구현을 작동시키는 것입니다. 그래서 qnode에서도 ros 구현 측면에서 관련된 포인터를 내부 멤버변수로 삼아 코딩을 하는 것이 qt-qnode-ros 구조 상 더 자연스러워보입니다.

이를 구현하기 위해 기존의 3번에서 했던 코드들을 qnode.hpp, qnode.cpp에서도 사용하고자 했습니다. QNode라는 클래스가 정의되어 있고 추가적으로 해당 파일에 publisher 클래스를 기입하여 QNode에게 publisher클래스의 shared_ptr을 멤버변수로 넣어줘서 main_window에서 qnode를 통해 publisher에 접근하여 동작을 하게 하는 것이 큰 틀입니다.

우선 publisher에는 publish역할을 해줄 pub포인터와 Client역할을 해줄 set_pen 포인터들을 생성해주었습니다.

```cpp
#include <rclcpp/rclcpp.hpp>
//3번 과제에서 인클루드 했던 대로 인클루드함
#include <geometry_msgs/msg/twist.hpp>
#include <turtlesim/srv/set_pen.hpp>
#include <std_srvs/srv/empty.hpp>
#endif
#include <QThread>

namespace Ui { class MainWindow; }
/*****************************************************************
** Class
*****************************************************************
class publisher: public rclcpp::Node{ //publisher 클래스 선언
    //3번 과제에서 선언한 포인터 그대로 사용
    rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr pub;
    rclcpp::Client<turtlesim::srv::SetPen>::SharedPtr set_pen;

    public:
    Ui:: MainWindow* ui; //ui에 접근하기 위해 포인터 선언하고 publisher 내부
    publisher();
    void publish_message(int flag); //버튼을 누르는 것으로 어떤 동작을 수행하
};
```

그리고 Ui::MainWindow* ui 를 선언해주었습니다. 그 이유는 cmd/vel 값을 gui출력해야 했기 때문입니다. main_window 파일에서 실시간으로 앞으로 이동, 회전 등에 대한 정보를 실시간으로 전달하려면 publish_message()내부에서 ui 포인터를 통해 바로 윈도우창에 바로 출력시키는 것이 제일 코드적으로 깔끔하고 편한 방법이었습니다. 그 이전에는 정보를 실시간으로 전달하기 위한 방법으로 subscribe_node 클래스를 만들어서 main_window에서 전달된 publish정보를 같은 토픽명으로 정보를 받아오는 것을 구상하여 실제 코딩해보았지만 결국 메시지를 받았을 때 호출할 콜백함수에서 내부적으로 ui포인터를 선언하여 사용해야 된다는 것을 알게 되어 그렇게 복잡하게 작동시키는 것보다 차라리 publisher클래스에서 이를 구현하는 것이 더 나은 방법이라고 생각이 들었습니다. 그래서 이 publisher 클래스에서 ui포인터를 선언하여 사용하게 된 것입니다. 아래의 사진은 subscribe_node를 통해 시도하다가 실패한 설계에 대한 코드를 나타낸 것입니다. 그리고 main_window파일에서 메시지를 받아 정보를 얻고 MainWindow 클래스의 slot함수에서 이를 처리하려 했기에 main_window에서 선언, 구현하려고 했었습니다.

```
// class subscribe_node: public rclcpp::Node{
//    rclcpp::Subscription<geometry_msgs::msg::Twist>::SharedPtr sub;
//    Ui:: MainWindow* ui;
//    void topic_callback(const geometry_msgs::msg::Twist::SharedPtr msg);

//    public:
//    subscribe_node(Ui::MainWindow* ui);
// };
#endif  // hw2_pkg_MAIN_WINDOW_H
```

```
// subscribe_node::subscribe_node(Ui::MainWindow* ui):Node("subscribe_node"), ui(ui){
//     sub = this->create_subscription<geometry_msgs::msg::Twist>("turtle1/cmd_vel",10,std::bind(&subscribe_node::topic_callbac
// }
// void subscribe_node:: topic_callback( const geometry_msgs::msg::Twist::SharedPtr msg){
//     ui->lineEdit-> setText("linear: "+QString::number(msg->linear.x) + "," + "angular: "+QString::number(msg->angular.z));
// }
```

```
public:
Ui:: MainWindow* ui; //ui에 접근하기 위해 포인터 선언하고 publisher 나
publisher();
void publish_message(int flag); //버튼을 누르는 것으로 어떤 동작을 수
```

그리고 publish_message라는 함수를 만들어서 어떤 버튼이 누르냐에 따라 동작을 달리하게 해야 했기에 main_window의 slot함수에서 매개변수 값을 다르게 주어 각기 flag값에 따라 다른 메시지를 publish하도록 만든 것입니다.

```
class QNode : public QThread
{
  Q_OBJECT
public:
  QNode();
  ~QNode();
  std::shared_ptr<publisher> pub_ptr;  //Qt쪽인 main_window와 ros간의
                                       //그래서 main_window의 qnode포인
protected:
  void run();

private:


Q_SIGNALS:
  void rosShutDown();
};

#endif /* hw2_pkg_QNODE_HPP_ */
```

QNode 클래스를 선언하는 부분입니다. 여기서 기본적인 설정 위에 shared_ptr로 publisher 객체를 둬서, 말씀드린 대로, main_window에서 qnode 포인터를 통해 publisher를 작동시키고자 이를 public 멤버변수로 선언하였습니다.

(2) qnode.cpp 파일 설명

```cpp
#include "../include/hw2_pkg/qnode.hpp"
#include "ui_mainwindow.h" //ui 접근 위해 인클루드
#include <QCoreApplication>
#include <QEventLoop>
#include <chrono>
QNode::QNode()
{
  int argc = 0;
  char** argv = NULL;
  rclcpp::init(argc, argv);
  pub_ptr = std::make_shared<publisher>();  //publisher 클래스 생성
  this->start();
}

QNode::~QNode()
{
  if (rclcpp::ok())
  {
    rclcpp::shutdown();
  }
}

void QNode::run()
{
  rclcpp::WallRate loop_rate(20);
  while (rclcpp::ok())
  {
    rclcpp::spin_some(pub_ptr);
    loop_rate.sleep();
  }
  rclcpp::shutdown();
  Q_EMIT rosShutDown();
}
```

ui의 lineEdit에 접근하기 위해서는 ui파일에 접근해야 하고 이를 코드로 접근하려면 ui_mainwindow.h이 필요하기에 이를 인클루드 했습니다. QNode의 구현부는 추가한 부분이 pub_ptr이므로 많은 구현 작업이 필요하지 않았습니다. make_shared로 객체를 생성하는 것과 그리고 spin을 하며 계속 돌아가게 해야 하므로 rclcpp의 spin을 하는 부분에 pub_ptr를 넣는 작업을 했습니다. 이는 기존 ros만 구동시킬 때의 main.cpp에서 하는 것을 QNode에서 하는 것 같습니다. 이 패키지의 main.cpp는 main_window와 관련된 작업을 하는 구조로 되어 있기 때문입니다.

```cpp
publisher::publisher():Node("publish_node"){ // publisher 클래스 생성자
    pub = this->create_publisher<geometry_msgs::msg::Twist>("turtle1/cmd_vel",10);
    set_pen = this->create_client<turtlesim::srv::SetPen>("turtle1/set_pen");

}
void publisher::publish_message(int flag){ //flag에 따라 어떤 도형을 그릴지 혹은 지울지 결정


    if(flag==1){ //정사각형
        auto req = std::make_shared<turtlesim::srv::SetPen::Request>();
        req->r = 255;
        req->g = 0;
        req->b = 0;
        req->width = 3;
        set_pen->async_send_request(req);
        geometry_msgs::msg::Twist temp;
        for(int i=0;i<4;i++){
        temp.linear.x=2.0;
        temp.angular.z=0.0;
        pub-> publish(temp);
        ui->lineEdit-> setText("linear: "+QString::number(temp.linear.x) + "," + "angular: "+
        QCoreApplication::processEvents(QEventLoop::AllEvents, 1);
        std::this_thread::sleep_for(std::chrono::seconds(3));
        temp.linear.x=0.0;
        temp.angular.z=1.57;
        pub-> publish(temp);
        ui->lineEdit-> setText("linear: "+QString::number(temp.linear.x) + "," + "angular: "+
        QCoreApplication::processEvents(QEventLoop::AllEvents, 1);
        std::this_thread::sleep_for(std::chrono::seconds(3));
    }
    }
```

이제 publisher의 생성자, publish_message를 구현하는 부분입니다. 생성자에서는 과제 3번과 같이 포인터만 create헬퍼 함수로 생성해주면 됩니다.

publish_message에서는 기존의 무한 반복인 while문을 뺐습니다. 그 이유는 QNode에서 spin이 돌아가고 있고 버튼이 눌리는 이벤트가 생길 때마다 publish_message가 호출되므로 마치 1초마다 publish되는 것과 비슷한 방식이기 때문입니다. 그리고 flag변수를 통해 메시지를 다르게 publish하도록 만들었습니다.

그 외에 추가된 것은 ui에 직접 접근하여 lineEdit에 setText로 해당 정보를 출력하는 것입니다. 이렇게 publish하고 곧바로 출력하게 하였습니다. 여기서 문제는 sleep과 ui의 동시 작동입니다. ui->lineEdit의 setText를 하고 sleep을 넣게 되면 거북이는 움직이지만 윈도우 창의 lineEdit에 텍스트가 출력되지는 않았습니다. 이는 ui가 setText를 완료하기 전에 sleep을 하여 setText하는 작동이 멈춘 것으로 보여졌기에 이를 해결할 방법을 서칭하였습니다. ros 구동과 ui구동을 동시에 해줄 것이 필요했습니다. 그래서 두 개의 동작을 동시에 해줄 수 있는 라이브러리를 찾던 중에 https://1d1cblog.tistory.com/292 링크의 사이트를 통해 QCoreApplication의 기능들을 통해 처리되지 못한 작업들을 해준다는 것을 알게 되었습니다.

void QCoreApplication::processEvents(QEventLoop::ProcessEventsFlags *flags*, int *ms*) [static]

This function overloads processEvents().

Processes pending events for the calling thread for *ms* milliseconds or until there are no more events to process, whichever is shorter.

You can call this function occasionally when your program is busy doing a long operation (e.g. copying a file).

Calling this function processes events only for the calling thread.

> **Note:** Unlike the processEvents() overload, this function also processes events that are posted while the function runs.

> **Note:** All events that were queued before the timeout will be processed, however long it takes.

Note: This function is thread-safe.

See also exec(), QTimer, and QEventLoop::processEvents().

그리고 이 기능이 제일 적합했기에 이를 사용한 것입니다. 이를 통해 sleep으로 간다고 하더라도 QCoreApplication 설정을 해놓았기에 버튼을 누르자마자 lineEdit에 출력이 바로 되는 것을 볼 수 있었습니다. 또한 이 메서드는 QEventLoop의 flag 변수를 사용하므로 qt의 QEventLoop의 documents로 들어가 모든 스레드에 동작하도록 설정하여 어떤 경우든 동시 동작이 가능하도록 만들었습니다.

```
else if(flag==2){ //정삼각형
    auto req = std::make_shared<turtlesim::srv::SetPen::Request>();
    req->r = 0;
    req->g = 255;
    req->b = 0;
    req->width = 7;
    set_pen->async_send_request(req);
    geometry_msgs::msg::Twist temp;
    for(int i=0;i<3;i++){
    temp.linear.x=2.0;
    temp.angular.z=0.0;
    pub-> publish(temp);
    ui->lineEdit-> setText("linear: "+QString::number(temp.linear.x) + ",
    QCoreApplication::processEvents(QEventLoop::AllEvents, 1);
    std::this_thread::sleep_for(std::chrono::seconds(3));
    temp.linear.x=0.0;
    temp.angular.z=2.09;
    pub-> publish(temp);
    ui->lineEdit-> setText("linear: "+QString::number(temp.linear.x) + ",
    QCoreApplication::processEvents(QEventLoop::AllEvents, 1);
    std::this_thread::sleep_for(std::chrono::seconds(3));
}
```

정삼각형을 그리는 경우도 이와 같습니다. ui로 LineEdit을 처리하고 QCoreApplication을 사용하는 등의 같은 방식을 사용했습니다.

(3) main_window.hpp/cpp파일 설명

```
#include <QMainWindow>
#include "qnode.hpp"
#include <geometry_msgs/msg/twist.hpp>

/*****************************************************************************
** Interface [MainWindow]
*****************************************************************************/
/**
 * @brief Qt central, all operations relating to the view part here.
 */

namespace Ui {
class MainWindow;
}
class MainWindow : public QMainWindow
{
  Q_OBJECT

public:
  MainWindow(QWidget* parent = nullptr);
  ~MainWindow();

  friend class publisher; //특정 버튼을 눌렀을 때 publish된 값을 바로 publis
private slots:
//각 버튼에 따른 클릭 슬롯함수
  void on_drawSquare_clicked();
  void on_drawCircle_clicked();
  void on_drawTriangle_clicked();
  void on_Delete_clicked();
private:
  Ui::MainWindow* ui;
  void closeEvent(QCloseEvent* event);
  QNode* qnode;

};
```

main_window의 hpp파일입니다. qnode.hpp를 선언하여 QNode를 사용하고자 하였고 패키지의 내부적인 부분이었던 qnode.hpp/cpp를 가지고 외부적인 부분에 해당하는 ui를 제어하기 위한 설계인 것을 볼 수 있습니다. 기본적인 ui, qnode포인터 등을 선언하였고 버튼이 눌렸을 때 publish를 해야 하므로 그에 대한 slots함수를 선언하였습니다. 여기서 중요한 부분은 friend 선언을 했다는 것입니다. friend 선언을 함으로써 publisher 클래스는 MainWindow클래스의 private, protected 멤버들을 publisher클래스 구현 내에서 자유자재로 부르거나 이용할 수 있게 됩니다. 그렇게 한 이유는 MainWindow클래스의 멤버

변수인 버튼 ui이나 lineEdit에 쉽게 접근할 수 있어야 하기에 이렇게 불가피하게 friend로 접근하게 되었습니다.

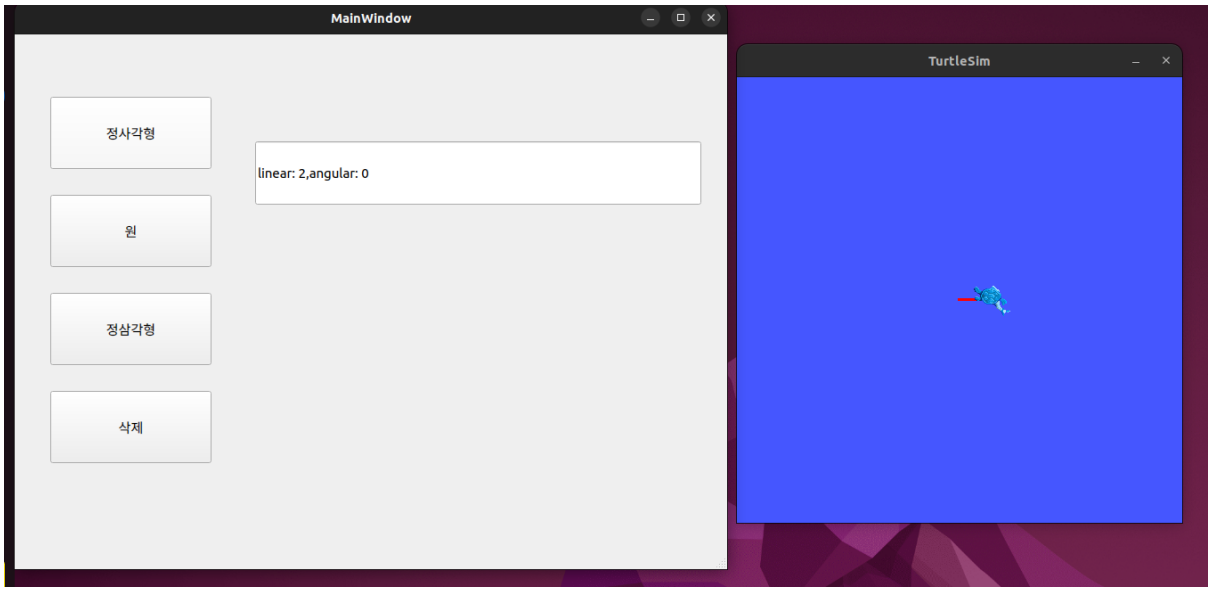
```
MainWindow::MainWindow(QWidget* parent) : QMainWindow(parent), ui(new Ui::MainWindow){
  ui->setupUi(this);



  qnode = new QNode();
  qnode->pub_ptr->ui = ui; //publisher 클래스의 ui포인터에 mainwindow의 ui포인터를 넣어줘서 pub

  QObject::connect(qnode, SIGNAL(rosShutDown()), this, SLOT(close()));

}

void MainWindow::closeEvent(QCloseEvent* event)
{
  QMainWindow::closeEvent(event);
}

MainWindow::~MainWindow()
{
  delete ui;
  delete qnode;

}
```

이제 main_window.cpp 구현파일입니다. ui를 setup하여 자동으로 signal과 slot을 연결해주는 등의 초기화 작업을 해줬습니다. 그리고 QNode 생성자로 qnode가 가리키는 객체가 생성되고 그 객체가 생성됨에 따라 publisher객체가 생성되게 됩니다. 그래서 이 생성자를 통해 모든 것이 한 번에 연결되는 구조가 되는 것입니다. 그리고 publisher의 ui 멤버변수에 대한 초기화를 해줘야 publish_message에서 알맞은 윈도우창을 제어할 수 있기 때문에 ui를 대입해줬습니다.

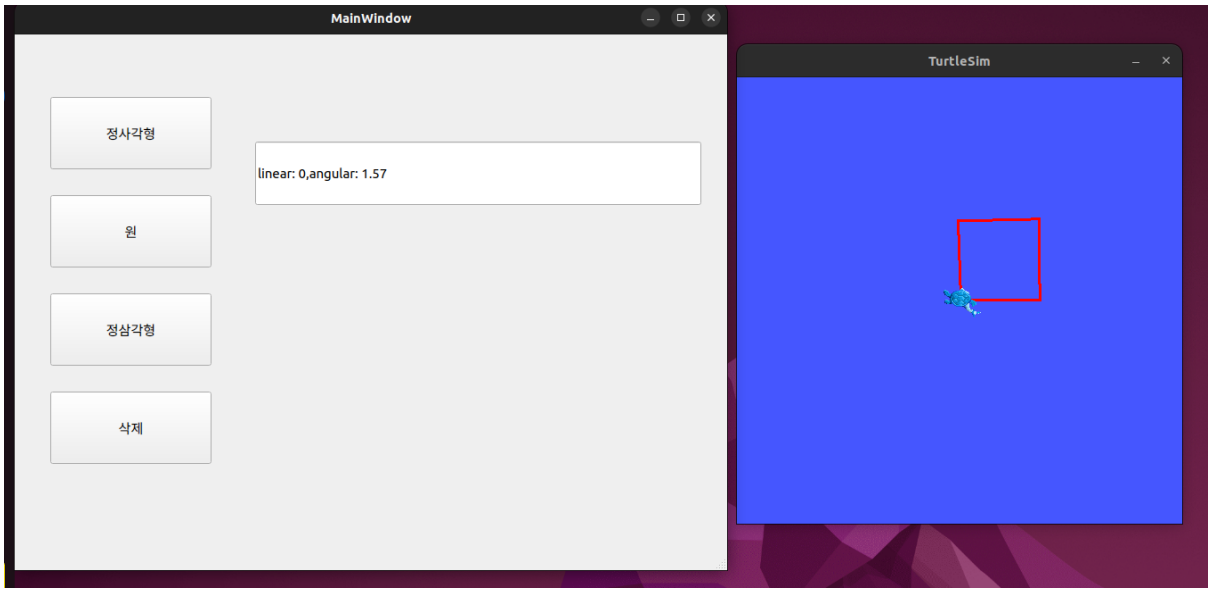
```
//각 번호에 대한 출력을 다르게 함 qnode의 publisher클래스의 publish_message함수를 호출
void MainWindow::on_drawSquare_clicked(){
  qnode-> pub_ptr->publish_message(1);
}
void MainWindow::on_drawTriangle_clicked(){
  qnode-> pub_ptr->publish_message(2);
}
void MainWindow::on_drawCircle_clicked(){
  qnode-> pub_ptr->publish_message(3);
}
void MainWindow::on_Delete_clicked(){
  qnode-> pub_ptr->publish_message(4);
}
```

그리고 버튼을 눌렀을 때 버튼에 따라 다른 메시지를 publish하기 위해 flag변수를 각기 다르게 주며 버튼과 대응되게 할당한 것입니다.

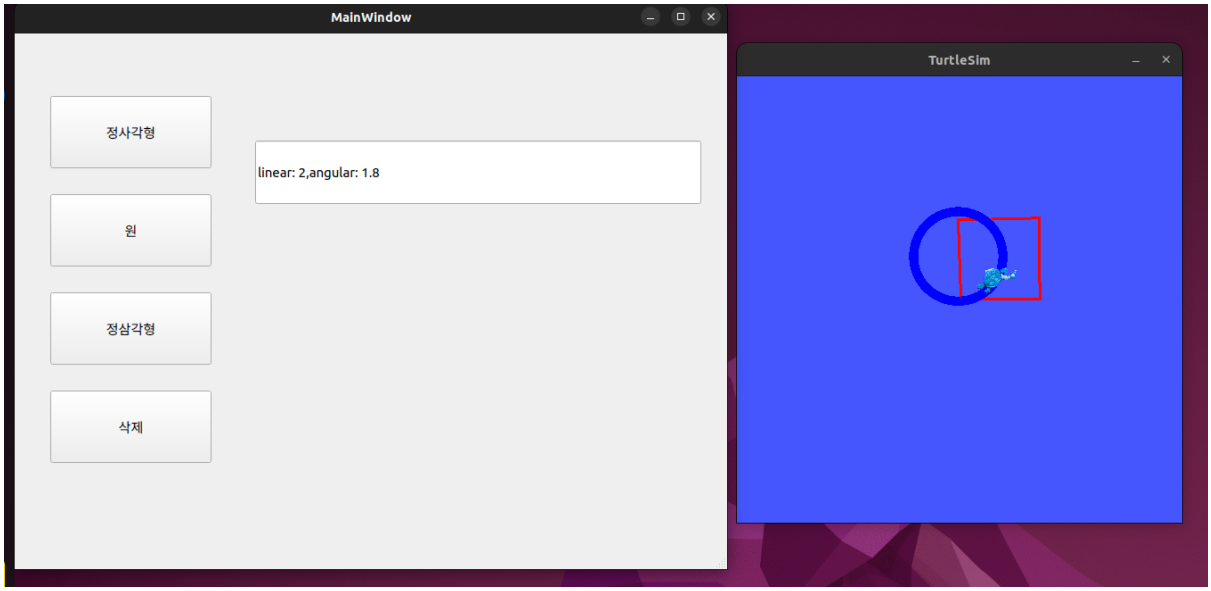
마지막으로 main.cpp는 기존의 qt를 했을 때의 main과 일치합니다.



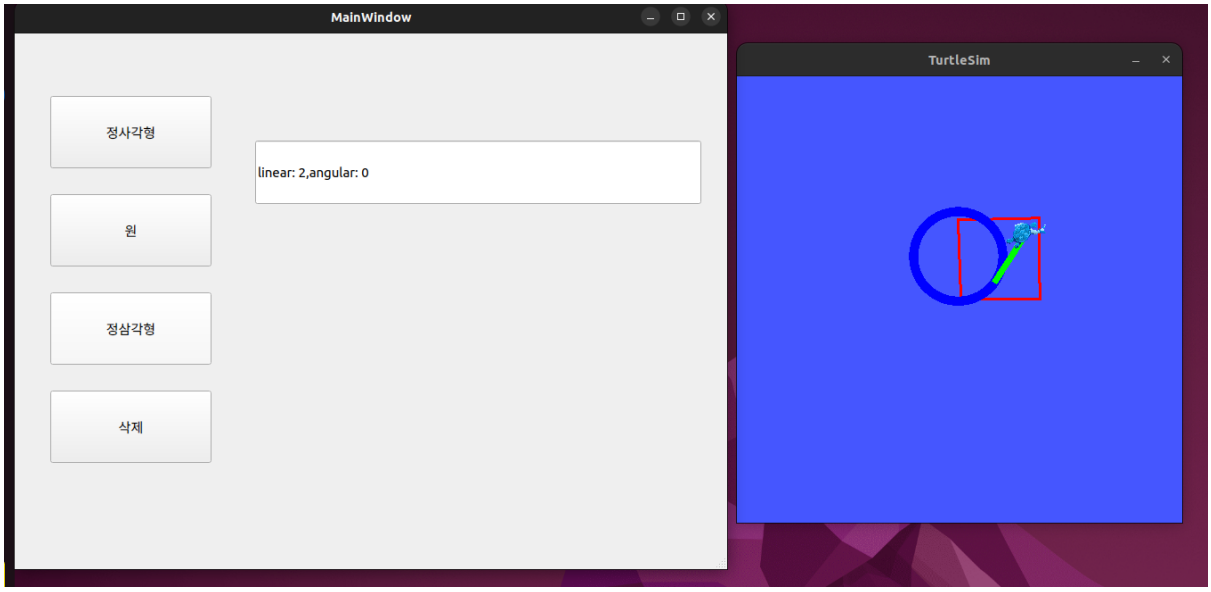
위 화면은 실행화면으로써 현재 정사각형 버튼을 누른 것이고 앞으로 진행하고 있는 순간을 찍은 것입니다. 현재 lineEdit에 linear, angular정보가 뜨고 있습니다.



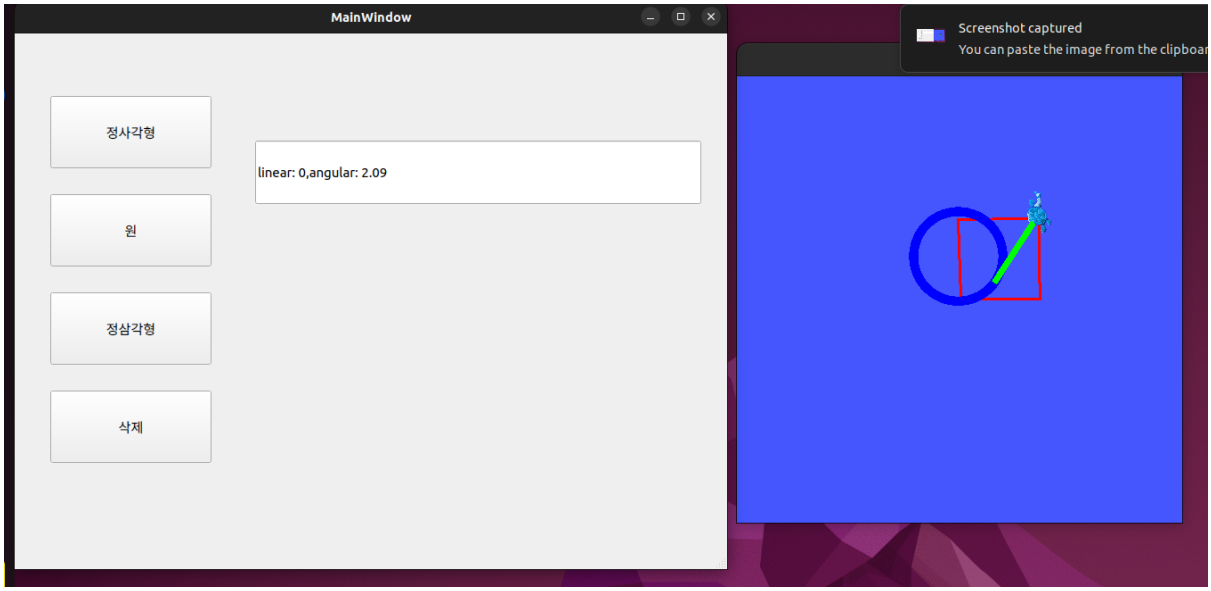
그리고 도는 순간에는 이렇게 linear, angular 정보가 다르게 출력됨을 보실 수 있습니다.



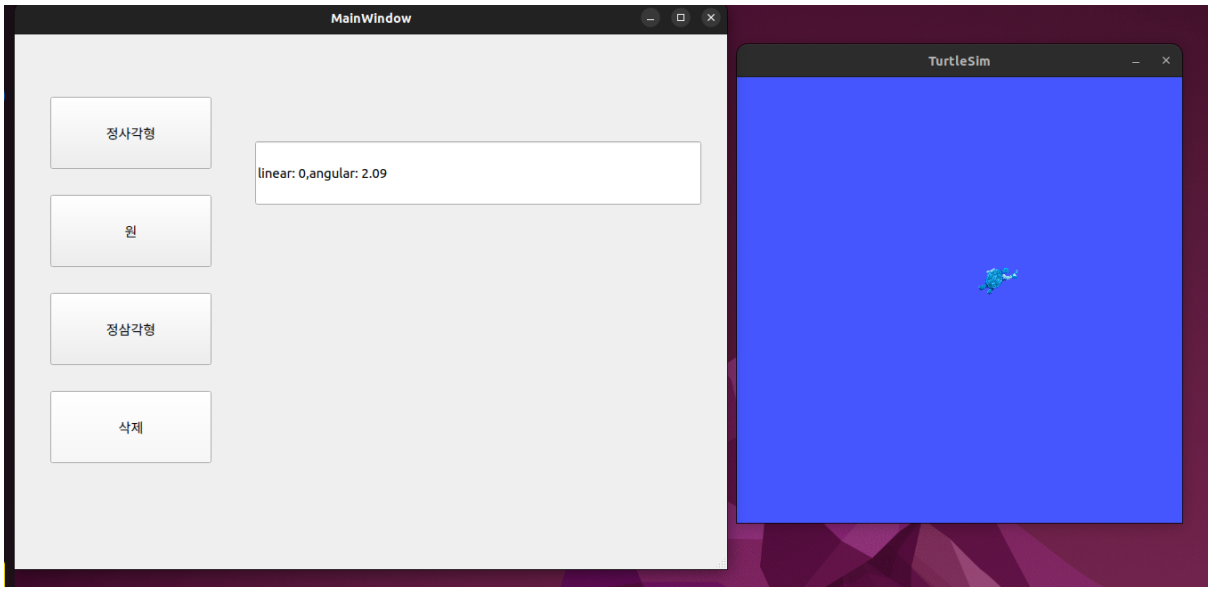
이번에는 원을 그리는 부분입니다.



삼각형을 그리는 부분입니다. 현재 진행하고 있는 상태의 장면을 찍은 것으로써 linear가 2, angular가 0으로 찍힌 상황입니다.



그리고 진행 후에 회전할 때의 장면을 찍은 것으로 lineEdit값이 다르게 나옴을 보실 수 있습니다.



그리고 삭제 버튼을 눌러서 지금까지 모두 그린 것을 지운 것입니다.