# Technical Design Documentation

## Overview

The Turtle Graphics application is a tool used for visualizing vector graphics patterns by manipulating a turtle cursor. The application is capable of reading scripts loaded from files or short user inputs directly from the nested command line interface (CLI). Patterns are then drawn on a canvas by executing commands in a sequence.

It is possible to save and load turtle states from files, allowing for version control. Additionally, GUI snapshots can be saved in the PNG format if needed.

Lastly, a gamification touch is added to the application by implementing obstacles diversifying the user experience. Thus, if the turtle cursor hits obstacles, the collision event is processed, possibly leading to engaging interactions.

## Software structure

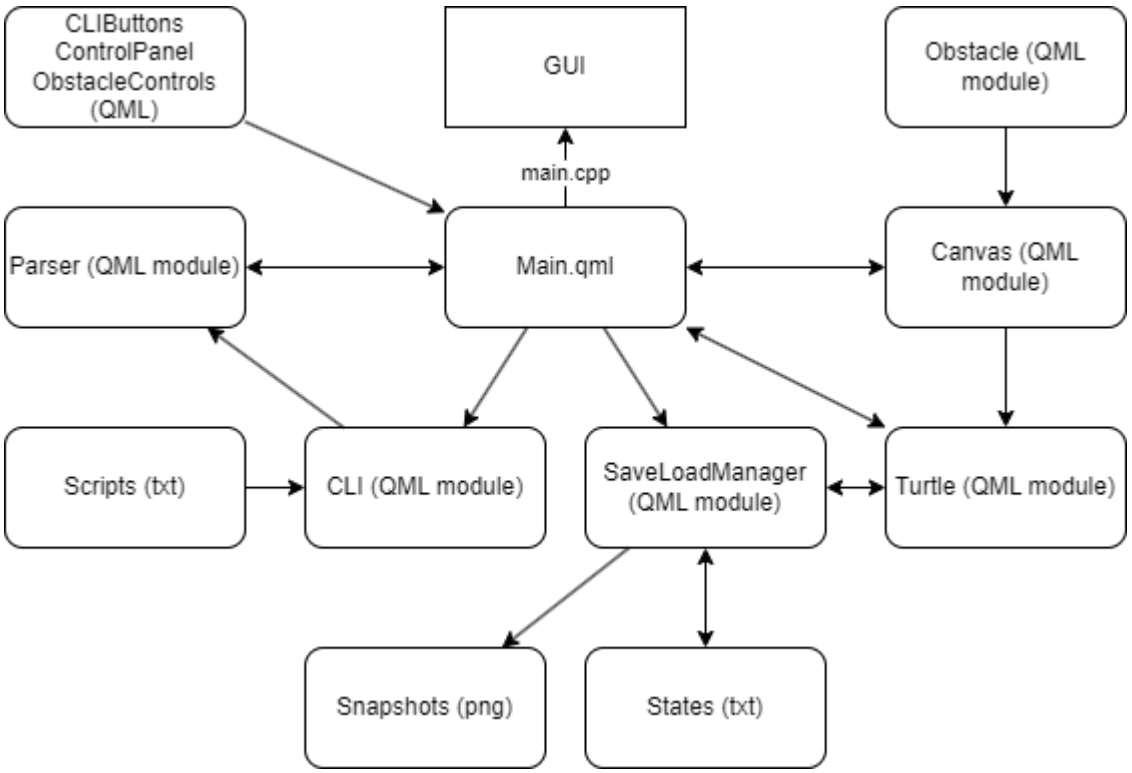



Figure 1. Control and data flow diagram of the application.

The application is developed using Qt Quick libraries. Its architecture follows a modular structure, where each module is responsible for managing a single feature at most. In this structure, *Main.qml* serves as a central hub linking QML modules with each other. This is

achieved by utilizing Qt signals and slots, as well as *Q_INVOKABLE* functions. Below is presented a code snippet from Main.qml, where the Parser's *animation_done* slot is connected to the *TurtleControl*'s *on_movement_completed* signal:

```
Connections {
  target: turtleControl
  function onOn_movement_completed(movement_result) {parser.animation_done()}
}
```

Additionally, the modules are linked by passing pointers during the initialization, allowing for direct communication between them:

```
Component.onCompleted: {
  cli.setParser(parser);
  turtleControl.set_canvas(canvasControl);
  stateManager.setTurtleControl(turtleControl);
  stateManager.setMainWindow(mainWindow);
}
```

# GUI

The GUI is designed using QtQuick and QML. Here is an overview of its key components:

- **QtQuick Canvas:** A drawable area where the turtle movements and obstacles are displayed. The canvas uses JavaScript for convenience to draw the objects. The canvas utilizes connections to refresh when the turtle state or obstacles change.
- **Bottom Control Panel:** Pen width and color controls and a reset canvas button. The control panel also integrates the CLI into the window via a command input field.
- **Feedback Display:** The turtle's position, rotation, pen color and the obstacle count is shown at the top of the screen. There is also an output panel at the bottom which provides command execution feedback.
- **Save/Load Features:** Buttons for saving a state or a screenshot or loading state/script.
- **Obstacle Controls:** Buttons for generating and clearing obstacles.

# Obstacle + Canvas

The Obstacle module defines an obstacle object that can be placed on the canvas. It encapsulates the shape, color, and position of the obstacle, and provides methods to manipulate these properties. The obstacle also has functionality to check if it intersects with a given rectangle, which is useful for collision detection.

The Canvas module manages the virtual canvas area where obstacles are placed and interacts with the turtle cursor. It allows for creating, positioning, and clearing obstacles within the canvas, as well as adjusting the canvas size. The properties of the canvas are defined and exposed to QML via the Qt property system.

# Parser

This module is responsible for parsing user inputs for valid commands and passing those commands to the Turtle module for execution. Scripting features like loops, functions and variables are also implemented here.

The **implementation** is simple and direct. For each command, a robust regular expression was written that matches both the command name and handles its arguments. When inputs are matched to a command regular expression, a corresponding Qt Signal is sent to the Turtle module, in order to execute the command.

Originally, the Parser was stateless, but to facilitate variable and function use, 3 unordered maps were added: one for storing variable values by name, one for storing the body-code of a function by function names, and one for helping pass the correct arguments to functions. The Parser waits for the Turtle module to finish executing its tasks before sending the next command; a boolean flag is used to do this. So the Parser has these **4 stateful elements**.

The **main function** in this class is parse_line which when given a string:

a. Splits the given line of text into command strings (splits with the ';' character)
b. Tries to match each command string with a regular expression. If the regular expression is matched, the corresponding Qt signal is emitted to the Turtle with the given arguments.
c. Returns a vector of the successfully parsed valid commands

This same function is used inside the function parse_script, which reads a file stream line-by-line. Additionally, parse_script has separate logic for function and loop definitions.

# CLI

The `CLI` class provides a command-line interface for interacting with the application. It processes commands input by user through UI, manages command history as well passes commands and script to the `Parser` class.

## Core Functionalities

- **Process Commands**: Interacts with the `Parser class` to interpret and execute commands input by user through UI.
- **Manage History**: Maintains a history of commands input by user during session for reference.
- **Output Logging**: Logs the results and outputs of commands for user reference.

## Implementation Details

### Command Processing

The `CLI` class delegates command parsing and execution to the `Parser` class:

- **processCommand(const QString &command)**: Sends a command string to the `Parser` and handles the response.
- **loadScript(const QString &filename)**: Reads a script file and passes its contents to the `Parser` for execution.

**Command History**

Maintains a history of commands for easy referencing and navigation:

- `QList<QString> commandHistory_`: Stores the list of commands entered by the user.
- `int historyIndex_`: Tracks the current position in the command history.

**Output Management**

Manages and provides access to the output logs:

- `QStringList outputLog_`: Stores log messages resulting from command execution.
- **getOutput() const**: Retrieves the current output log as a single string.
- **clearOutput()**: Clears the output log.

## Key Methods

- **setParser(Parser *parser)**: Pass the pointer of the `Parser instance to SaveLoadManager class` for command processing.
- **processCommand(const QString &command)**: Processes a given command or string of command input by user and passes it to `Parser` .
- **getOutput() const**: Retrieves the current output log.
- **clearOutput()**: Clears the current output log.
- **getCommandHistory() const**: Returns the history of executed commands as a `QStringList`.
- **loadScript(const QString &filename)**: Loads and processes a script file through the `Parser`.

## Signals

- **commandProcessed(const QString &message)**: Emitted when a command has been processed.
- **outputChanged()**: Emitted when the output log changes.
- **requestQuit()**: Emitted to request the application to quit.

## Members

- `Parser *parser_`: Pointer to the `Parser` instance for command processing.
- `QList<QString> commandHistory_`: List of user-entered commands.
- `int historyIndex_`: Current position in the command history.
- `QStringList outputLog_`: Log of command outputs.

This class provides a robust interface for command entry and processing, integrating seamlessly with the `Parser` class and ensuring a responsive and user-friendly command-line experience.

## TurtleControl

The *TurtleControl* QML module holds the turtle cursor's data and handles its movement in the canvas, physical interactions with other objects, and pen actions.

Properties are communicated with the help of the Qt property system. For instance, the cursor position property is exposed to the QML language and is defined as follows:

```
Q_PROPERTY(QPointF position READ position WRITE set_position NOTIFY position_changed FINAL)
```

The cursor movement is executed by animating the cursor properties using the *Animation Framework* (*QPropertyAnimation* and *QParallelAnimationGroup* classes):

```
// Animating the forward movement
QPropertyAnimation *anim = new QPropertyAnimation(this, "position", this);
anim->setDuration(duration);
anim->setStartValue(position_);
anim->setEndValue(new_position);
anim->start()
```

Physical interactions, such as collisions with obstacles and canvas borders, are handled within the *anim_value_changed* function. In case if the cursor hits an object, the movement is blocked and the *on_collision* signal is emitted allowing for further processing of the event, such as adding visual effects or making the cursor bounce from surfaces.

```
void TurtleControl::anim_value_changed()
{
        if (!current_anim_) {
        return;
        }

        if (!test_collision()) {
        update_lines();
        previous_position_ = position_;
        previous_rotation_ = rotation_;
```

```
        }
}
```

## SaveLoadManager

The `SaveLoadManager` class manages the state of the application, including saving/loading state as well as capturing screenshots of the application window.

### Core Functionalities

- **Save Application State**: Serializes and writes the current state to a file in the form of a string of different arguments separated by comma..
- **Load Application State**: Reads and deserializes state from a file to restore the application.
- **Capture Screenshots**: Saves the current view of the main window in the form .png image file.

### Key Methods

- **setMainWindow(QObject *mainWindow)**: Sets the main window reference.
- **setTurtleControl(TurtleControl *turtleControl)**: Sets the `TurtleControl` reference.
- **setBuildFolder(const QString &buildFolder)**: Defines the folder for saving/loading files.
- **saveScreenshot()**: Captures and saves a screenshot of the main window.
- **saveState(const QString &fileName)**: Saves the state to a file with name specified through save dialogue..
- **loadState(const QString &filePath)**: Loads state from a file select through file dialogue.

### Private Helpers

- **saveToFile(const QString &filePath, const QString &content)**: Writes string of turtle and line arguments to a file.
- **loadFromFile(const QString &filePath)**: Reads content from a file which contains the saved state.
- **getCurrentDateTimeString() const**: Generates a timestamp string for filenames of screenshot.

### Members

- `QObject *m_mainWindow`: Reference to the main window.
- `TurtleControl *m_turtleControl`: Reference to the `TurtleControl`.
- `QString m_buildFolder`: Path for saving/loading files.

This concise and efficient design allows `SaveLoadManager` to effectively manage state and integrate seamlessly with QML, enhancing the user experience and application management.

# Build Instructions

## Windows

### Using Qt Creator

1. Open the project by loading *CMakeLists.txt* in the root directory
2. Configure project kits (Qt Creator should automatically open the corresponding dialog when loading the project for the first time)
3. Build with the Debug configuration

### Using terminal

1. Ensure that Qt and CMake are installed
2. Set up environment variables for Qt, for example:
    - *set CMAKE_PREFIX_PATH=C:\Qt\6.8.0\msvc2022_64*
    - *set Qt6_DIR=C:\Qt\6.6.3\msvc2019_64\lib\cmake\Qt6*
3. Open the build directory
4. Configure the project with *cmake ..*
5. Build the project with *cmake --build .*

### Build Instructions Using CMake on Linux

Follow these steps in order to build the project using `CMake` on Linux.

#### Prerequisites

Ensure you have the following installed on your system before running the application:

- Qt (Version 6.6)
- CMake (Version 3.5 or later)
- GCC or Clang Compiler

### Step-by-Step Build Instructions

1. **Prepare Your Environment** Ensure Qt and CMake are already installed on your system. If not then you can install them using your package manager by typing following in terminal:

    ```
    sudo apt-get install qtcreator
    ```

2. **Set Qt Path in CMakeLists.txt** In your `CMakeLists.txt` file, add the path to your Qt installation with the `CMAKE_PREFIX_PATH` variable. For example:

```
set(CMAKE_PREFIX_PATH "${CMAKE_PREFIX_PATH};/path/to/Qt/6.x.x/gcc_64")
```

3. **Generate Makefile with CMake** Create a build directory and then navigate into it:

```
mkdir build && cd build
```

4. Run the following command to generate the Makefile using `CMake`:

```
cmake ..
```

5. **Build the Project with make** Once the Makefile is generated, build the project using `cmake --build .` :

```
cmake --build .
```

### Running the Application

After successful building of the project, you will have an executable file in your build directory. Run the executable by navigating to the build directory and executing the following command in terminal:

```
./ProjectName
```

By following these instructions, you can build and run the project on Linux using CMake and make. Remember to adjust the paths in the CMakeLists.txt file to match your local setup if you change the project setup..

### Build Instructions for Project Using Qt Creator on Linux

Follow these simple steps in order to build the project using Qt Creator on Linux.

### Prerequisites

Ensure you have the following installed on your system before-hand:

- Qt (Version 6.6 recommended)
- Qt Creator

If Qt Creator is not installed on your system, you can simply install it by typing the following in the terminal:

```
sudo apt-get install qtcreator
```

### Step-by-Step Build Instructions

1. **Open Qt Creator**
    - Launch Qt Creator from your applications menu or either by typing `qtcreator` in the terminal. Make sure you have properly set the environment variable for Qt Creator in order to run it from the terminal.
2. **Open Project**
    - Click on `File` in the top menu.
    - Select `Open File or Project...`.
    - Navigate to your project directory and select the `CMakeLists.txt` file.
    - Click `Open`.
3. **Configure Project**
    - Qt Creator will prompt you to configure the project.
    - Select the appropriate build kit for your environment (e.g., `Desktop Qt 6.x.x GCC 64bit`).
    - Click `Configure Project`.
4. **Build the Project**
    - Click on the `Build` button (hammer icon) or either you can build the project by pressing `Ctrl+B` to start the build process.
    - Qt Creator will automatically run CMake and build the project.
5. **Run the Application**
    - Once the build is completed, click on the `Run` button (green play icon) or either press `Ctrl+R` to run your application.
    - The application will execute, and you can see the UI window for project application.

By following these instructions, you can easily build and run the project using Qt Creator on Linux with just a few clicks.

# User Guide

**BASIC USE:**

The simplest way to use this app is to write commands on the embedded command line. The basic commands are as follows: (here x and y stand in for floating point numbers)

- forward(x)
    - Move the turtle x steps forward
- turn(x)
    - Turn x degrees clockwise
- setpos(x, y)
    - Sets the position of the Turtle on the Canvas
- setrot(x)
    - Sets the orientation of the Turtle relative to North in degrees
- setsize(x)
- setspeed(x)

- arc(x,y)
    - Draws an arc of radius x, sweeping an angle y (degrees)
- up
    - Stop drawing with the Turtle, though it can still be moved normally
- down
    - Resume drawing with the Turtle

Multiple commands can be given on the same line by separating them with the “;” character.

**ADVANCED USE:**

**Buttons:**

- Generate Obstacles: Creates 3 random obstacles on the canvas. Avoids overlapping with the turtle.
- Clear Obstacles: Removes all obstacles from the canvas.
- Save: Saves the current canvas state to a file. Obstacles not included.
- Load: Loads a previously saved canvas state.
- Screenshot: Takes a snapshot of the current canvas view and saves it.
- Load Script: Loads a script for automated turtle movement.
- Reset Canvas, pen width and pen color: Self explanatory.

**Advanced CLI use:**

VARIABLES

- Float type variables can be defined by writing for example “x=8.8” or “y=-2”. A variable “new” can be made from an existing variable “old” by writing “new=old”.
- There are two arithmetic operations for manipulating variables: these are “z=add(x,y)” and “z=mul(x,y)”. Here x and y can be float values or the names of existing variables

FINITE SINGLE-LINE LOOPS

- A command line can be repeated an integer N times by writing it inside the {}-brackets in the expression “LOOPN{}”
- E.g. “LOOP4{forward(100);turn(90)}” draws a square

**Scripts:**

A series of commands can be written to a file, where commands are separated by new-lines or “;” characters. In addition, scripts enable user-defined functions. Nested loops or functions (ie. loops inside loops, functions inside functions, loops inside functions and vice versa) are **not** supported.

LOOPS IN SCRIPTS

- In addition to the single-line loops, the loop body can be on multiple lines in a script, but the beginning “LOOPN{“ and the closing bracket “}” should be on their own lines.

FUNCTIONS

- Functions can pass a single argument to a code-block consisting of any commands except for loops or other function definitions.

- As with loops, the DEF line and the closing brackets need to be on otherwise empty lines.
- E.g. “DEF func(x){“ is a valid DEF line (the x can also be omitted if no argument is needed)
- User-defined functions can currently have at most one argument. E.g. “DEF f(x,y){“ does not work.

# Testing

C++ files in the QML modules (*TurtleControl*, *Canvas*, and *Obstacle*) are tested using Qt Tests. A separate test project is created in the /test directory and linked as a subdirectory (*add_subdirectory(tests/testturtle*) in *CMakeLists.txt* to the main app. Since testing is enabled and tests are added in CMake (*add_test(NAME TestTurtle COMMAND TestTurtle)*), the test cases are automatically executed during project builds:

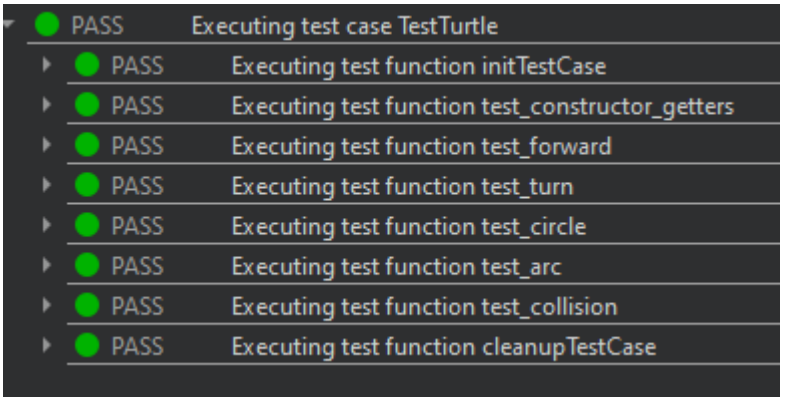
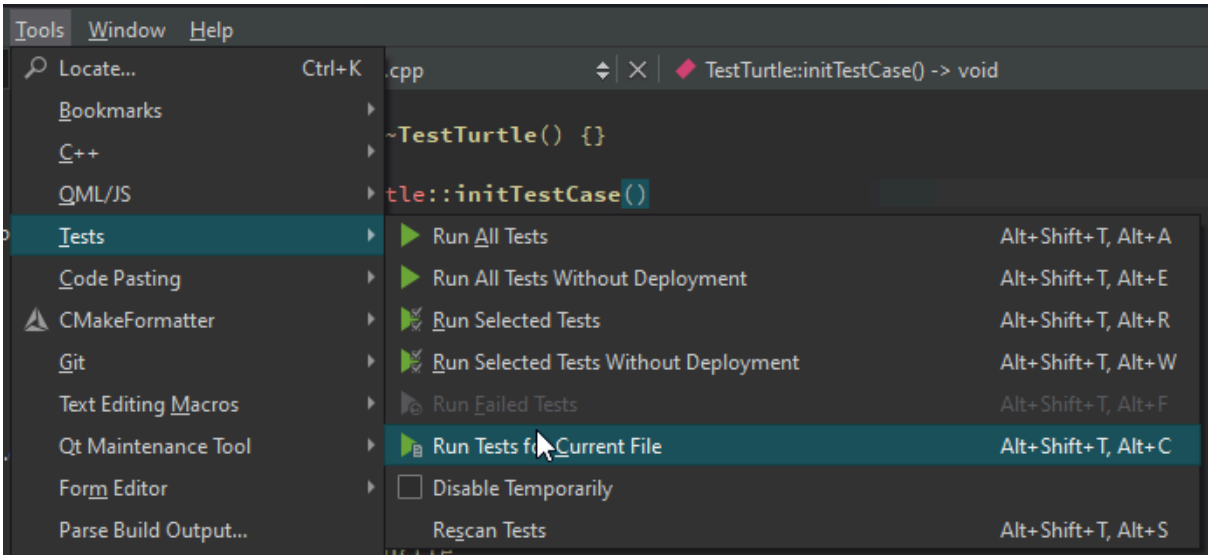



Figure 2. TurtleControl test cases passed all verifications.

Alternatively, the tests can be performed manually via terminal or using the Qt Creator GUI:



# TurtleControl

The *TurtleControl* module includes test cases for each movement command and additionally checks collisions with obstacles.

First, TurtleControl and Canvas objects are created during the test initialization:

```
void TestTurtle::initTestCase()
{
        canvas_ = new Canvas();
        canvas_->set_width(800.0);
        canvas_->set_height(600.0);
        turtle_ = new TurtleControl();
        initial_turtle_position_ = turtle_->position();
        turtle_->set_canvas(canvas_);
}
```

Then, test cases are executed in order and verified using *QVERIFY* and *QCOMPARE* macros. For instance, the forward command can be tested as following:

- Bind a *QSignalSpy* to the on_movement_completed signal
- Execute the forward command
- Wait for a signal emission
- Very returned values

```
// Set up a spy and call the command
QSignalSpy spy(turtle_, &TurtleControl::on_movement_completed);
QVERIFY(spy.isValid());
const float duration = turtle_->forward(distance);
spy.wait(duration * 1.1f);

QCOMPARE(spy.count(), 1); // making sure the signal was emitted exactly one time
QList<QVariant> arguments = spy.takeFirst(); // take the first signal
QVERIFY(!arguments.isEmpty());
result = (MovementResult) arguments.at(0).value<MovementResult>();

QCOMPARE(result, MovementResult::kSuccess);
QCOMPARE(turtle_->position().toPoint(), expected_position.toPoint());
```

## Parser

Once the ability to read commands from a script was implemented, these scripts were used to test the parser. A simple fuzzer was written to generate randomized valid command-scripts. A list of valid commands was written from which commands were randomly sampled to the test script, in order to ensure that strange sequences of commands don't cause issues. The fuzzer could also inject whitespace and special characters into commands. The fuzzer was used to discover issues, but the core of the testing was simply manual use during development.

Since the parser simply uses regular expressions to match for valid commands and ignores erroneous commands, it suffices to test positively that valid commands work. Erroneous commands can **not** cause app-breaking bugs due to the design of the Parser. The parser was tested to work with the various example scripts as well as through the CLI.

## SaveLoadManager Test

The `test_SaveLoadManager` class is designed to verify the functionality of the `SaveLoadManager` class in saving and loading the turtle state. This test involves setting up and initializing instances of `SaveLoadManager` and `TurtleControl` classes, and configuring the `TurtleControl` with specific attributes such as position, rotation, pen state, pen radius, and pen color. After configuration, this test involves saving the state of the turtle using `SaveLoadManager` in .txt file format. The saved state from the .txt file is subsequently loaded back and passed to `TurtleControl`, and the test verifies that the loaded state matches the original state. Finally, the test includes cleanup of the saved state file and the created directory, ensuring a thorough and efficient validation of the save and load functionality.

1. **Setup and Initialization**:
   - Instances of SaveLoadManager and TurtleControl classes are created.
   - The SaveLoadManager is configured to manage the TurtleControl.
   - A directory test_build_folder is created if it doesn't exist.

2. **State Configuration**:
   - The `TurtleControl` is set with specific attributes such as:
   - Position: (50, 50)
   - Rotation: 45 degrees
   - Pen state: down
   - Pen radius: 2
   - Pen color: red (#ff0000)

   These attributes are set using the respective methods:

   - turtleControl.set_position(position);
   - turtleControl.set_rotation(rotation);
   - turtleControl.set_pen_down(penDown);

- turtleControl.set_pen_radius(penRadius);
- turtleControl.set_pen_color(penColor);

3. **State Saving**:
   The current state of `TurtleControl` is saved to a .txt file named `test_state` utilizing the `SaveLoadManager`.

4. **State Resetting**:
   The current state of the `TurtleControl instance` is reset to default values to ensure that loading mechanism is being tested effectively:

- Position: `(0, 0)`
- Rotation: `0 degrees`
- Pen state: up
- Pen radius: 1
- Pen color: `black`

5. **State Loading**:

   The saved state is loaded back into `TurtleControl` from the file.

6. **Verification**:

   The loaded state is verified to match the original saved state using `QCOMPARE`:

```
QCOMPARE(turtleControl.position(), position);
QCOMPARE(turtleControl.rotation(), rotation);
QCOMPARE(turtleControl.pen_down(), penDown);
QCOMPARE(turtleControl.pen_radius(), penRadius);
QCOMPARE(turtleControl.pen_color(), penColor);
```

7. **Outcome of test:**


# Valgrind test:

The memory leak test output specifically mentions following memory errors:

1. **Still Reachable Memory Errors**

    - **16 bytes in 1 block still reachable**:

    `QtSharedPointer::ExternalRefCountData::getAndRef(QObject const*)` as used in various modules (Turtle, Parser, CLI, Obstacle, Canvas, SaveLoadManager).

    These occurrences can be seen for various modules and point to the fact that the `QObject` pointers that should be parent-managed and their life cycle managed by parent QObject are still reachable and not properly cleaned up upon exit of application.

    - **24 bytes in 1 block still reachable**:
    - For plugin instance creation in various modules (Turtle, Parser, CLI, Obstacle, Canvas, SaveLoadManager).
    - **32 bytes in 1 block still reachable**:
    - In `main` function in `/src/main.cpp`.
    - **120 bytes in 1 block still reachable**:
    - During instantiation of module plugins (TurtleModulePlugin, ParserModulePlugin, CLIModulePlugin, ObstacleModulePlugin, CanvasModulePlugin, SaveLoadManagerModulePlugin).

Each of these headers and memory blocks are associated predominantly within Qt's framework during plugin instantiation and use in the main application.

# Work log

The work log is exported from the Gitlab Issue Board and is provided as a separate file in */doc*.