Documentation of 461 Drawing Project

Longwood Spring 2021 CS Senior Seminar Students

March 31, 2021

Contents

0.1 Preface

This compilation of documentation exists as a reference for students in the CMSC 461 Senior Seminar course at Longwood University. The Following code has been created by all the students of the class, and an attempt has been made to make a cross platform application that is useful and simple, and document it in a detailed manner.

System Dependencies

1.0.1 For Linux:

To get QT:

-

To get C++ & GCC:

-

To get Python3:

-

1.0.2 For Windows:

To get QT:

-

To get C++ & GCC:

-

To get Python3:

-

1.0.3 For MacOS:

To get QT:

- Go to https://www.qt.io click on "Download. Try."
- Scroll down and click on "Go open source"
- Scroll down and click on "Download the Qt Online Installer"
- Click "Download"

To get C++ & GCC:

- Open the terminal app
- Type g++ <filename>
- if prompted download mac's develper tools

To get Python3:

- Open terminal type: /bin/bash -c

  ```
  "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install.sh)"
  ```

- At the bottom of your " /.profile" file type:

  ```
  export PATH="/usr/local/opt/python/libexec/bin:$PATH"
  ```

- type "brew install python"

JSON Protocol

Defined here is what a JSON Object should look like for sending and receiving shapes.

For circles:

```
{
    "id": 1,
    "shape": "circle",
    "data": {
        "radius": "10",
        "center": {
            "x": 100,
            "y": 150
        }
    }
}
```

For rectangles:

```
{
    "id": 2,
    "shape": "line",
    "data": {
        "start": {
            "x": 10,
            "y": 50
        },
        "end": {
            "x": 150,
            "y": 100
        }
    }
}
```

Client

The following is an explanation of the client side of the 461 Drawing Project

ItemStats

3.0.1 .h

itemStats is a struct which contains all of the values which a shape/object contains. This is used in other documents and functions to help parse out and separate data.

```
struct itemStats
{
        //Lines get tracked the same as everything else.
        //Set the width/height variables to x2 and y2, respectively.
        std::string board_id;
        std::string type;
        int id;
        double x;
        double y;
        double height;
        double width;
        QColor rgb;

        //Functions to turn the thing into a QJsonObject
        //and a QByteArray that can be sent directly over a socket.

  QJsonObject toJson();
        QByteArray byteData();

        //Several different constructors for a couple of different
        //uses. You can pass in nothing, a QGraphicsItem* for the
        //client end, or a bunch of variables that it will simply
        //insert.
        itemStats();
        itemStats(std::string board_id, QGraphicsItem* item);
        itemStats(std::string board_id, std::string type, int id, double x, double y, \
  double height, double width, QColor rgb);
  ~itemStats();
};
```

3.0.2 .cpp

```
itemStats::itemStats(std::string nboard_id, std::string ntype, int nid, double nx, double ny, \
double nheight, double nwidth, QColor nrgb)
  //UPDATE THIS IF YOU ADD THINGS TO THE PLACED ITEMS

itemStats::itemStats(std::string nboard_id, QGraphicsItem* item)
  /*UPDATE THIS IF YOU ADD THINGS TO THE PLACED ITEMS
  Parses the QGraphicsItem and throws the variables into their appropriate places.*/

itemStats::~itemStats()
  //destructor for itemStats

QJsonObject itemStats::toJson()
  /*UPDATE THIS IF YOU ADD THINGS TO THE PLACED ITEMS
  Turns the item into a JSON object, formatted as it would be
  in the jsonExamples folder. This doesn't actually differ at
  all from different shape types.

  For the 'end' parameter, it uses the width and height as x and y coordinates.
  If the thing's a line, the end is just what it says on the tin - the ending coordinates of the line. */

QByteArray itemStats::byteData()
  /*Turns it first into a JSON object, then passes that object
        into a QByteArray. This array can be sent straight to the server.*/
```

ProjectScene

3.0.3 .h

projectScene is a class which contains functions that relate to the QGraphicsScene, that is, the 'behinds the scenes' of the graphical window.

```
class ProjectScene : public QGraphicsScene
{
        Q_OBJECT
        private:
                QTcpSocket* m_socket;
                std::vector<itemStats> m_tracked_items;
                std::string m_board_id;

        public slots:
                void sceneChanged(const QList<QRectF> &region);
                void readSocket();
                void disconnect();
        signals:
        public:
                int trackItem(QGraphicsItem* item);
                ProjectScene();
                ~ProjectScene();
};
```

3.0.4 .cpp

```
ProjectScene::ProjectScene()
  //A constructor for the ProjectScene class

ProjectScene::~ProjectScene()
  //A destructor for the ProjectScene class

ProjectScene::readSocket()
  //Reads in data that is received from the server into a JSON object

ProjectScene::disconnect()
  /*Deletes the socket from the scene, effectively removing the connection from
    the server.*/

ProjectScene::trackItem()
  /*Pushes back all the data from a received graphics item into an internal list
    of graphics items.*/

ProjectScene::sceneChanged()
  /*When something in the scene changes, this event is triggered. What we do is
   get a list of items that have changed (compare between the last update and
   our internal list) and then tells the server about it.*/
```

ProjectView

3.0.5 .h

projectView is a class which contains the functions which relate to the QGraphicsView, that is, the canvas and buttons that we see on the graphical window.

```
class ProjectView : public QGraphicsView
{
        Q_OBJECT
        private:
                int m_tool;
                int m_color_r;
                int m_color_g;
                int m_color_b;

        QPointF firstClick;
        private slots:

        public:
                void change_tool(int tool);
                void change_color(int r, int g, int b);

        void circle_tool(qreal x, qreal y,qreal x2, qreal y2);
        void line_tool(qreal x, qreal y, qreal x2, qreal y2);
                void rect_tool(qreal x, qreal y, qreal x2, qreal y2);
                void mousePressEvent(QMouseEvent* event);
                void mouseReleaseEvent(QMouseEvent* event);
                ProjectView();
                ~ProjectView();
};
```

3.0.6 .cpp

```
ProjectView::ProjectView()
  //A constructor for the ProjectView class

ProjectView::~ProjectView()
  //A destructor for the ProjectView class

ProjectView::change_tool(int tool)
  //Change the current tool based on an integer parameter

ProjectView::change_color(int r, int g, int b)
    //Change the current tool's color based on 3 integer parameters, r, g, and b.

ProjectView::circle_tool(qreal x, qreal y, qreal x2, qreal y2)
  /*Draw a circle on the canvas using two points, (x,y) and (x2,y2). This allows
    us to draw ellipses easily as well as circeles, similarly to how we draw
    rectangles.
    (x,y) is the first location the user clicked, and (x2,y2) is the second.*/

ProjectView::line_tool(qreal x, qreal y, qreal x2, qreal y2)
    /*Draw a line on the canvas using two points, (x,y) and (x2,y2).
    (x,y) is the first location the user clicked, and (x2,y2) is the second.*/

ProjectView::rect_tool(qreal x, qreal y, qreal x2, qreal y2)
    /*Draw a rectangle on the canvas using two points, (x,y) and (x2,y2).
    (x,y) is the first location the user clicked, and (x2,y2) is the second.*/

ProjectView::mousePressEvent()
     /*This event is triggered when the user clicks down on the mouse. Technically,
     it stores this point and doesn't do anything with it until a mouseReleaseEvent.*/

ProjectView::mouseReleaseEvent()
    /*This event is triggered when the user releases the click on the mouse. This
    function takes the second point, figures out which action to take based on
    the selected tool, and calls the appropriate function.*/
```

ToolBar

3.0.7 .h

toolbar is a class which contains the functions to change the different tools in the graphical window.

```
class ToolBar : public QWidget
{
        Q_OBJECT
        private:
                QPushButton* m_circle;
                QPushButton* m_line;
                QPushButton* m_rect;
                QPushButton* m_default;
                QPushButton* m_black;
                QPushButton* m_red;
                QPushButton* m_green;
                QPushButton* m_yellow;
```

```
        QPushButton* m_blue;
        QPushButton* m_color_picker;
        QVBoxLayout* m_layout;
        ProjectView* m_view;
    public slots:
        // set default
        void set_default();
        // shapes
        void place_rectangle();
        void set_line();
        void set_circle();
        // colors
        void set_color_black();
        void set_color_red();
        void set_color_green();
        void set_color_yellow();
        void set_color_blue();
        void set_color_custom();

    public:
        ToolBar();
        ~ToolBar();
        void set_view(ProjectView* view);

};
```

3.0.8 .cpp

```
ToolBar::ToolBar()
  //A constructor for the ToolBar class

ToolBar::~ToolBar()
  //A destructor for the ToolBar class

ToolBar::set_view(ProjectView* view)
  //Sets the tool's m_view to view

ToolBar::set_default()
  //Sets the default tool to the select tool

ToolBar::set_line()
  //Sets the tool to the line tool

ToolBar::set_circle()
  //Sets the tool to the circle tool

ToolBar::set_rect()
  //Sets the tool to the rect tool

ToolBar::set_color_black()
  //Sets the tool's color to be black

ToolBar::set_color_red()
  //Sets the tool's color to be red

ToolBar::set_color_green()
```

```
  //Sets the tool's color to be green

ToolBar::set_color_yellow()
  //Sets the tool's color to be yellow

ToolBar::set_color_blue()
  //Sets the tool's color to be blue



ToolBar::set_color_custom()
  /*Brings up a color picker menu for the user to set their own color using a
    graphical interface, rgb, hexadeicimal, or cmyk.*/
```

Window

3.0.9 .h

window is essentially the 'main' of the graphical window, though there is a 'main' that spawns it.

```
class Window : public QMainWindow //Extension on the base QMainWindow class
{
        Q_OBJECT //Must be included for qmake to recognize this

        private: //Menus and features of the window
                ToolBar* m_bar;
                QDockWidget* m_tool_dock;
                ProjectScene* m_scene;
                ProjectView* m_view;

        // shape tools

        private slots: //Where functions attached to buttons go
        // make a popup window
                void popup();
        public:
                Window();
                ~Window();
};
```

3.0.10 .cpp

```
Window::Window()
  /*A constructor for the Window class. Sets up the entire graphical window with
    toolbars, docks, a canvas, a scene, and a menu.*/

Window::~Window()
  //A destructor for the Window class.
```

Main

3.0.11 .cpp

main.cpp is a small file which contains one function: int main. This is the start of the entire client, where a new window is created in memory and executed. From there, upon returning, the event loop begins.

```
int main(int argc, char* argv[])
  /*This is the start of the entire client, where a new window is created in memory and executed.*/
```

Server

The server is run off of a Qt framework using Qt sockets. Clients can connect to the server via these Qt sockets, and are assigned unique identifiers using the socketDescriptor value of the connecting socket. By maintaining a constant connection, the server is able to receive and send the most current information about databases and shapes.

Server

4.0.1 .h

Server.h includes some tools to set up an item based on its data and turn it into a byte array. Also turn it into a QJsonObject.

```
struct ownedDB{
    int id;
    QSqlDatabase db;
} newDB;

class Server : public QMainWindow
{
        Q_OBJECT
        private:
                QTcpServer* m_server;
                QSet<QTcpSocket*> connected;
                std::string m_board_id;
        QVector<ownedDB> databases;

        public slots:
                void newConnection();
                void readSocket();
                void disconnect();
        public:
                Server();
                ~Server();
                void appendSocket(QTcpSocket* sock);
        void createBoard(QTcpSocket* socket);
        void deleteDB(QTcpSocket* socket);
};
```

4.0.2 .cpp

```
Server::Server() : QMainWindow()
  /*Sets up the server itself.
  The server will give out a 'signal' when it receives a new connection.
```

```
  This connects that signal to the function 'newConnection()'.*/

void Server::newConnection()
  /*Checks for pending connections, then calls the other function that actually adds the socket
        to the list of sockets we have.*/

void Server::appendSocket(QTcpSocket* socket)
  //adds a new socket to all current clients are stored in the vector.

void Server::disconnect()
  //Black voodoo magic with a cast to determine which socket it was that needs to be disconnected.

void Server::readSocket()
  /*There is a QByteArray array of bytes that the socket has just sent to the server.
  The array is formatted as a JSON object. The client sends out this JSON object in
  the exact format that is specified in the jsonExamples folder. The data gets received and is
  ready to be parsed and turned into a string and outputs it to console. Casts the byte array
  into a Qt style JSON object. You can get keys and values out of this the same way
        you would anything else. Qt documentation will have more specific info on:*/

  QJsonDocument doc = QJsonDocument::fromJson(buf);
        QJsonObject obj = doc.object();

void Server::createBoard(QTcpSocket* socket)
      /*Create a QSqlDatabase, either connect to the database named CMSC461.db or
      create it if it doesn't exist. Create and QSqlQuery pointer to be used to execute commands*/

void Server::deleteDB(QTcpSocket* socket)
```

4.0.3 .py

```
  def getJsonFile():

  def addCircle(bid,sid,r,x,y,color):
    #Creates JSON-formatted circle data

  def addLine(bid,sid,x1,y1,x2,y2,color):
    #Creates JSON-formatted line data

  def addRect (bid,sid,x,y,w,h,color):
    #Creates JSON-formatted rectangle data

  @app.route('/addShape', methods = ['POST', 'GET'])
  def shapeType():
    #Checks shape type received and calls matching function

  @app.route('/saveFile', methods = ['POST', 'GET'])
  def saveFile():
    # A implementation of saveFile which allows the user to save the board that they are currently using.
    # User will send this url to the server and server will reply back with an SVG file

  @app.route('/fullUpdate', methods = ['POST', 'GET'])
  def fullUpdate():
    """This function keeps track of all shapes across all clients concurrently via a QByteArray of JSON objects. In order to

  @app.route('/createBoard', methods = ['POST', 'GET'])
```

```
  def createBoard():
    # Create a new database for every new board that is created
```

Main

4.0.4 .cpp

```
int main(int argc, char* argv[])
```