MainWindow.cpp

#include "mainwindow.h"

#include "ui\_mainwindow.h"

#include "ModelPart.h"

#include "ModelPartList.h"

#include "optiondialog.h"

#include <QMessageBox>

#include <QFileDialog>

#include <functional>

#include <vtkCylinderSource.h>

#include <vtkPolyDataMapper.h>

#include <vtkActor.h>

#include <vtkProperty.h>

#include <vtkRenderer.h>

#include <vtkRenderWindow.h>

#include <vtkGenericOpenGLRenderWindow.h>

#include <vtkCamera.h>

#include <vtkSmartPointer.h>

MainWindow::MainWindow(QWidget \*parent)

: QMainWindow(parent)

, ui(new Ui::MainWindow)

{

ui->setupUi(this);

// Initialize ModelPartList

this->partList = new ModelPartList("PartsList");

ui->treeView->setModel(this->partList);

// Add items to the tree

ModelPart\* rootItem = this->partList->getRootItem();

for (int i = 0; i < 3; i++) {

QString name = QString("TopLevel %1").arg(i);

QString Colour = "R";

QString visible("true");

ModelPart\* childItem = new ModelPart({name, visible});

rootItem->appendChild(childItem);

for (int j = 0; j < 5; j++) {

QString name = QString("Item %1,%2").arg(i).arg(j);

QString colour("R");

QString visible("true");

ModelPart\* childChildItem = new ModelPart({name, visible});

childItem->appendChild(childChildItem);

}

}

//WS6 Ex10

ui->treeView->setContextMenuPolicy(Qt::ActionsContextMenu);

QAction\* actionItemOptions = new QAction(tr("Item Options"), this);

ui->treeView->addAction(actionItemOptions);

// Connect the tree view click signal to the slot

connect(ui->treeView, &QTreeView::clicked, this, &MainWindow::handleTreeClicked);

connect(ui->pushButton, &QPushButton::released, this, &MainWindow::handleButton);

connect(ui->pushButton\_2, &QPushButton::released, this, &MainWindow::handleSaveButton);

//connect statusbar with message

connect(this, &MainWindow::statusUpdateMessage, ui->statusbar, &QStatusBar::showMessage);

// Connect the action signal to the slot

connect(ui->actionOpen\_File, &QAction::triggered, this, &MainWindow::on\_actionOpenFile\_triggered);

connect(actionItemOptions, &QAction::triggered, this, &MainWindow::on\_actionItemOptions\_triggered);

// Link a render window with the Qt widget

renderWindow = vtkSmartPointer<vtkGenericOpenGLRenderWindow>::New();

ui->vtkWidget->setRenderWindow(renderWindow);

// Add a renderer

renderer = vtkSmartPointer<vtkRenderer>::New();

renderWindow->AddRenderer(renderer);

// Create an object and add to renderer

vtkNew<vtkCylinderSource> cylinder;

cylinder->SetResolution(8);

vtkNew<vtkPolyDataMapper> cylinderMapper;

cylinderMapper->SetInputConnection(cylinder->GetOutputPort());

vtkNew<vtkActor> cylinderActor;

cylinderActor->SetMapper(cylinderMapper);

cylinderActor->GetProperty()->SetColor(1.0, 0.0, 0.35);

cylinderActor->RotateX(30.0);

cylinderActor->RotateY(45.0);

renderer->AddActor(cylinderActor);

// Reset camera

renderer->ResetCamera();

renderer->GetActiveCamera()->Azimuth(30);

renderer->GetActiveCamera()->Elevation(30);

renderer->ResetCameraClippingRange();

}

MainWindow::~MainWindow()

{

delete ui;

}

/\*Original handleButton

void MainWindow::handleButton() {

QMessageBox msgBox;

msgBox.setText("Add button was clicked");

emit statusUpdateMessage("Button was clicked", 2000); // Emit the signal

msgBox.exec();

}\*/

void MainWindow::handleButton() {

// This causes MainWindow to emit the signal that will then be

// received by the status bar’s slot

emit statusUpdateMessage(QString("Add button was clicked"), 500);//emit the message after each release and close by 500ms

}

void MainWindow::handleTreeClicked(const QModelIndex &index) {

// Get a pointer to the item from the index

ModelPart\* selectedPart = static\_cast<ModelPart\*>(index.internalPointer());

// Retrieve the name string from the internal QVariant data array

QString text = selectedPart->data(0).toString();

// Emit a status update message

emit statusUpdateMessage(QString("The selected item is: ") + text, 0);

}

void MainWindow::on\_actionOpenFile\_triggered() {

// Open file dialog

QString fileName = QFileDialog::getOpenFileName(

this,

tr("Open File"),

"C:\\Users\\Zhixing\\2024\_20291563\\worksheet6",

tr("STL Files (\*.stl);;Text Files (\*.txt)")

);

if (fileName.isEmpty()) {

return; // No file selected, exit function

}

// Extract only the file name, removing path

QFileInfo fileInfo(fileName);

QString shortFileName = fileInfo.fileName(); // e.g., "model.stl"

// Emit message showing the selected file name

emit statusUpdateMessage("Selected file: " + shortFileName, 5000);

// Get the currently selected item in the tree view

QModelIndex index = ui->treeView->currentIndex();

if (!index.isValid()) {

QMessageBox::warning(this, "Selection Error", "Please select an item in the tree.");

return;

}

// Update the item's name in the tree view

ModelPart\* selectedPart = static\_cast<ModelPart\*>(index.internalPointer());

if (selectedPart) {

selectedPart->set(0, shortFileName); // Update the name

// Notify the model that data has changed to refresh the UI

emit ui->treeView->model()->dataChanged(index, index, {Qt::DisplayRole});

}

}

void MainWindow::handleOptionDialog() {//WS6 Ex9

OptionDialog dialog(this);

if (dialog.exec() == QDialog::Accepted) {

QString inputText = dialog.getInputText();

emit statusUpdateMessage(QString("Dialog accepted. Input: ") + inputText, 5000);

} else {

emit statusUpdateMessage(QString("Dialog rejected"), 5000);

}

}

void MainWindow::on\_actionItemOptions\_triggered() {

QModelIndex index = ui->treeView->currentIndex();

if (!index.isValid()) {

QMessageBox::warning(this, "Selection Error", "Please select an item to edit.");

return;

}

ModelPart\* selectedPart = static\_cast<ModelPart\*>(index.internalPointer());

if (!selectedPart) {

return;

}

OptionDialog dialog(this);

dialog.setWindowTitle("Edit Item Properties");

// Set initial values in dialog

dialog.setInputText(selectedPart->data(0).toString()); // Name

dialog.setRGB(selectedPart->getColourR(), selectedPart->getColourG(), selectedPart->getColourB()); // Colour

dialog.setItemVisible(selectedPart->visible()); // Visibility

if (dialog.exec() == QDialog::Accepted) {

QString newName = dialog.getInputText();

int newR = dialog.getR();

int newG = dialog.getG();

int newB = dialog.getB();

bool newVisibility = dialog.isItemVisible();

QModelIndex visibilityIndex = index.sibling(index.row(), 2); // Get the visibility column index

// Update ModelPart

selectedPart->set(0, newName);

selectedPart->setColour(newR, newG, newB);

selectedPart->setVisible(newVisibility); // Update visibility in model

// Manually trigger `setData()` for visibility update

ui->treeView->model()->setData(visibilityIndex, newVisibility ? Qt::Checked : Qt::Unchecked, Qt::CheckStateRole);

// Force UI to refresh

emit statusUpdateMessage("Updated item: " + newName +

" (Colour: " + QString("%1,%2,%3").arg(newR).arg(newG).arg(newB) +

", Visible: " + (newVisibility ? "Yes" : "No") + ")", 5000);

ui->treeView->update();

}

}

//write to file

void MainWindow::saveTreeToFile(const QString &filePath) {

QFile file(filePath);//check and prepare to open file

if (!file.open(QIODevice::WriteOnly | QIODevice::Text)) {

QMessageBox::warning(this, "File Error", "Could not open file for writing.");

return;

}

QTextStream out(&file);

ModelPart\* root = partList->getRootItem();

// Recursive function to save items

std::function<void(ModelPart\*, int)> savePart = [&](ModelPart\* part, int depth) {//defining the function savePart, depth shows the levels

out << QString(depth \* 2, ' ')//write to file the related 3 columns

<< part->data(0).toString() << ","

<< part->getColourR() << "," << part->getColourG() << "," << part->getColourB() << ","

<< (part->visible() ? "true" : "false") << "\n";

for (int i = 0; i < part->childCount(); i++) {

savePart(part->child(i), depth + 1);

}

};

for (int i = 0; i < root->childCount(); i++) {

savePart(root->child(i), 0);

}//

file.close();//file closed after being created and written

emit statusUpdateMessage("Tree saved to " + filePath, 5000);//file saved

}

void MainWindow::handleSaveButton() {

QString filePath = QFileDialog::getSaveFileName(this, tr("Save Tree"), "", tr("Text Files (\*.txt)"));

if (!filePath.isEmpty()) {

saveTreeToFile(filePath); // Now we have a valid file path

}

}

void MainWindow::updateRender() {

renderer->RemoveAllViewProps();

updateRenderFromTree(partList->index(0, 0, QModelIndex()));

renderer->Render();

MainWindow.h

#ifndef MAINWINDOW\_H

#define MAINWINDOW\_H

#include <QMainWindow>

#include "ModelPart.h"

#include "ModelPartList.h"

#include "Optiondialog.h"

#include <vtkCylinderSource.h>

#include <vtkPolyDataMapper.h>

#include <vtkActor.h>

#include <vtkProperty.h>

#include <vtkRenderer.h>

#include <vtkRenderWindow.h>

#include <vtkGenericOpenGLRenderWindow.h>

#include <vtkCamera.h>

#include <vtkSmartPointer.h>

QT\_BEGIN\_NAMESPACE

namespace Ui {

class MainWindow;

}

QT\_END\_NAMESPACE

class MainWindow : public QMainWindow

{

Q\_OBJECT

public:

MainWindow(QWidget \*parent = nullptr);

~MainWindow();

public slots://for pushbutton1

void handleButton();

public slots://for pushbutton2Ex9

void handleOptionDialog();

public slots://Ex10

void on\_actionItemOptions\_triggered();

public slots:

void saveTreeToFile(const QString &filePath); // Declare the function

public slots:

void handleSaveButton(); // Intermediate slot

public slots:

void handleTreeClicked(const QModelIndex &index);//Ex5 for tree click

public slots://Ex6 for open file

void on\_actionOpenFile\_triggered();

signals:

void statusUpdateMessage(const QString &message, int timeout);

private:

Ui::MainWindow \*ui;

// This needs to be added to the MainWindow class definition

private:

vtkSmartPointer<vtkRenderer> renderer;

vtkSmartPointer<vtkGenericOpenGLRenderWindow> renderWindow;

private:

ModelPartList\* partList;//add for declare a pointer to partlist.

};

#endif // MAINWINDOW\_H

Main.cpp

#include "mainwindow.h"

/\*

#include "ui\_mainwindow.h"

#include "ModelPart.h"

#include "ModelPartList.h"

\*/

#include <QApplication>

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

{

QApplication a(argc, argv);

MainWindow w;

w.show();

return a.exec();

}

ModelPart.cpp

/\*\* @file ModelPart.cpp

\*

\* EEEE2076 - Software Engineering & VR Project

\*

\* Template for model parts that will be added as treeview items

\*

\* P Evans 2022

\*/

#include "ModelPart.h"

/\* Commented out for now, will be uncommented later when you have

\* installed the VTK library

\*/

//#include <vtkSmartPointer.h>

//#include <vtkDataSetMapper.h>

ModelPart::ModelPart(const QList<QVariant>& data, ModelPart\* parent )

: m\_itemData(data), m\_parentItem(parent) {

/\* You probably want to give the item a default colour \*/

}

ModelPart::~ModelPart() {

qDeleteAll(m\_childItems);

}

void ModelPart::appendChild( ModelPart\* item ) {

/\* Add another model part as a child of this part

\* (it will appear as a sub-branch in the treeview)

\*/

item->m\_parentItem = this;

m\_childItems.append(item);

}

ModelPart\* ModelPart::child( int row ) {

/\* Return pointer to child item in row below this item.

\*/

if (row < 0 || row >= m\_childItems.size())

return nullptr;

return m\_childItems.at(row);

}

int ModelPart::childCount() const {

/\* Count number of child items

\*/

return m\_childItems.count();

}

int ModelPart::columnCount() const {

/\* Count number of columns (properties) that this item has.

\*/

return m\_itemData.count();

}

QVariant ModelPart::data(int column) const {

/\* Return the data associated with a column of this item

\* Note on the QVariant type - it is a generic placeholder type

\* that can take on the type of most Qt classes. It allows each

\* column or property to store data of an arbitrary type.

\*/

if (column < 0 || column >= m\_itemData.size())

return QVariant();

return m\_itemData.at(column);

}

void ModelPart::set(int column, const QVariant &value) {

/\* Set the data associated with a column of this item

\*/

if (column < 0 || column >= m\_itemData.size())

return;

m\_itemData.replace(column, value);

}

ModelPart\* ModelPart::parentItem() {

return m\_parentItem;

}

int ModelPart::row() const {

/\* Return the row index of this item, relative to it's parent.

\*/

if (m\_parentItem)

return m\_parentItem->m\_childItems.indexOf(const\_cast<ModelPart\*>(this));

return 0;

}

void ModelPart::setColour(const unsigned char R, const unsigned char G, const unsigned char B) {

/\* This is a placeholder function that you will need to modify if you want to use it \*/

colourR = R;

colourG = G;

colourB = B;

/\* As the name suggests ... \*/

}

unsigned char ModelPart::getColourR() {

/\* This is a placeholder function that you will need to modify if you want to use it \*/

return colourR;

/\* As the name suggests ... \*/

return 0; // needs updating

}

unsigned char ModelPart::getColourG() {

/\* This is a placeholder function that you will need to modify if you want to use it \*/

return colourG;

/\* As the name suggests ... \*/

return 0; // needs updating

}

unsigned char ModelPart::getColourB() {

/\* This is a placeholder function that you will need to modify if you want to use it \*/

return colourB;

/\* As the name suggests ... \*/

return 0; // needs updating

}

void ModelPart::setVisible(bool visible) {

isVisible = visible;

}

bool ModelPart::visible() const {

return isVisible;

}

void ModelPart::loadSTL( QString fileName ) {

/\* This is a placeholder function that you will need to modify if you want to use it \*/

/\* 1. Use the vtkSTLReader class to load the STL file

\* https://vtk.org/doc/nightly/html/classvtkSTLReader.html

\*/

/\* 2. Initialise the part's vtkMapper \*/

/\* 3. Initialise the part's vtkActor and link to the mapper \*/

}

//vtkSmartPointer<vtkActor> ModelPart::getActor() {

/\* This is a placeholder function that you will need to modify if you want to use it \*/

/\* Needs to return a smart pointer to the vtkActor to allow

\* part to be rendered.

\*/

//}

//vtkActor\* ModelPart::getNewActor() {

/\* This is a placeholder function that you will need to modify if you want to use it

\*

\* The default mapper/actor combination can only be used to render the part in

\* the GUI, it CANNOT also be used to render the part in VR. This means you need

\* to create a second mapper/actor combination for use in VR - that is the role

\* of this function. \*/

/\* 1. Create new mapper \*/

/\* 2. Create new actor and link to mapper \*/

/\* 3. Link the vtkProperties of the original actor to the new actor. This means

\* if you change properties of the original part (colour, position, etc), the

\* changes will be reflected in the GUI AND VR rendering.

\*

\* See the vtkActor documentation, particularly the GetProperty() and SetProperty()

\* functions.

\*/

/\* The new vtkActor pointer must be returned here \*/

// return nullptr;

//}

ModelPart.h

/\*\* @file ModelPart.cpp

\*

\* EEEE2076 - Software Engineering & VR Project

\*

\* Template for model parts that will be added as treeview items

\*

\* P Evans 2022

\*/

#include "ModelPart.h"

/\* Commented out for now, will be uncommented later when you have

\* installed the VTK library

\*/

//#include <vtkSmartPointer.h>

//#include <vtkDataSetMapper.h>

ModelPart::ModelPart(const QList<QVariant>& data, ModelPart\* parent )

: m\_itemData(data), m\_parentItem(parent) {

/\* You probably want to give the item a default colour \*/

}

ModelPart::~ModelPart() {

qDeleteAll(m\_childItems);

}

void ModelPart::appendChild( ModelPart\* item ) {

/\* Add another model part as a child of this part

\* (it will appear as a sub-branch in the treeview)

\*/

item->m\_parentItem = this;

m\_childItems.append(item);

}

ModelPart\* ModelPart::child( int row ) {

/\* Return pointer to child item in row below this item.

\*/

if (row < 0 || row >= m\_childItems.size())

return nullptr;

return m\_childItems.at(row);

}

int ModelPart::childCount() const {

/\* Count number of child items

\*/

return m\_childItems.count();

}

int ModelPart::columnCount() const {

/\* Count number of columns (properties) that this item has.

\*/

return m\_itemData.count();

}

QVariant ModelPart::data(int column) const {

/\* Return the data associated with a column of this item

\* Note on the QVariant type - it is a generic placeholder type

\* that can take on the type of most Qt classes. It allows each

\* column or property to store data of an arbitrary type.

\*/

if (column < 0 || column >= m\_itemData.size())

return QVariant();

return m\_itemData.at(column);

}

void ModelPart::set(int column, const QVariant &value) {

/\* Set the data associated with a column of this item

\*/

if (column < 0 || column >= m\_itemData.size())

return;

m\_itemData.replace(column, value);

}

ModelPart\* ModelPart::parentItem() {

return m\_parentItem;

}

int ModelPart::row() const {

/\* Return the row index of this item, relative to it's parent.

\*/

if (m\_parentItem)

return m\_parentItem->m\_childItems.indexOf(const\_cast<ModelPart\*>(this));

return 0;

}

void ModelPart::setColour(const unsigned char R, const unsigned char G, const unsigned char B) {

/\* This is a placeholder function that you will need to modify if you want to use it \*/

colourR = R;

colourG = G;

colourB = B;

/\* As the name suggests ... \*/

}

unsigned char ModelPart::getColourR() {

/\* This is a placeholder function that you will need to modify if you want to use it \*/

return colourR;

/\* As the name suggests ... \*/

return 0; // needs updating

}

unsigned char ModelPart::getColourG() {

/\* This is a placeholder function that you will need to modify if you want to use it \*/

return colourG;

/\* As the name suggests ... \*/

return 0; // needs updating

}

unsigned char ModelPart::getColourB() {

/\* This is a placeholder function that you will need to modify if you want to use it \*/

return colourB;

/\* As the name suggests ... \*/

return 0; // needs updating

}

void ModelPart::setVisible(bool visible) {

isVisible = visible;

}

bool ModelPart::visible() const {

return isVisible;

}

void ModelPart::loadSTL( QString fileName ) {

/\* This is a placeholder function that you will need to modify if you want to use it \*/

/\* 1. Use the vtkSTLReader class to load the STL file

\* https://vtk.org/doc/nightly/html/classvtkSTLReader.html

\*/

/\* 2. Initialise the part's vtkMapper \*/

/\* 3. Initialise the part's vtkActor and link to the mapper \*/

}

//vtkSmartPointer<vtkActor> ModelPart::getActor() {

/\* This is a placeholder function that you will need to modify if you want to use it \*/

/\* Needs to return a smart pointer to the vtkActor to allow

\* part to be rendered.

\*/

//}

//vtkActor\* ModelPart::getNewActor() {

/\* This is a placeholder function that you will need to modify if you want to use it

\*

\* The default mapper/actor combination can only be used to render the part in

\* the GUI, it CANNOT also be used to render the part in VR. This means you need

\* to create a second mapper/actor combination for use in VR - that is the role

\* of this function. \*/

/\* 1. Create new mapper \*/

/\* 2. Create new actor and link to mapper \*/

/\* 3. Link the vtkProperties of the original actor to the new actor. This means

\* if you change properties of the original part (colour, position, etc), the

\* changes will be reflected in the GUI AND VR rendering.

\*

\* See the vtkActor documentation, particularly the GetProperty() and SetProperty()

\* functions.

\*/

/\* The new vtkActor pointer must be returned here \*/

// return nullptr;

//}

ModelPartList.cpp

/\*\* @file ModelPartList.h

\*

\* EEEE2076 - Software Engineering & VR Project

\*

\* Template for model part list that will be used to create the trewview.

\*

\* P Evans 2022

\*/

#include "ModelPartList.h"

#include "ModelPart.h"

ModelPartList::ModelPartList( const QString& data, QObject\* parent ) : QAbstractItemModel(parent) {

/\* Have option to specify number of visible properties for each item in tree - the root item

\* acts as the column headers

\*/

rootItem = new ModelPart( { tr("Part"), tr("Visible?") } );

}

ModelPartList::~ModelPartList() {

delete rootItem;

}

int ModelPartList::columnCount( const QModelIndex& parent ) const {

Q\_UNUSED(parent);

Q\_UNUSED(parent);

return 4; // Part, Colour, Visible;

}

bool ModelPartList::setData(const QModelIndex& index, const QVariant& value, int role) {

if (!index.isValid())

return false;

ModelPart\* item = static\_cast<ModelPart\*>(index.internalPointer());

if (index.column() == 2 && role == Qt::CheckStateRole) { // Visibility column

bool isChecked = (value.toInt() == Qt::Checked);

item->setVisible(isChecked); // Update model data

emit dataChanged(index, index, {Qt::CheckStateRole, Qt::DisplayRole}); // Force UI update

return true;

}

return false;

}

QVariant ModelPartList::data(const QModelIndex& index, int role) const {

if (!index.isValid())

return QVariant();

ModelPart\* item = static\_cast<ModelPart\*>(index.internalPointer());

if (role == Qt::DisplayRole) {

if (index.column() == 0) { // Part Name Column

return item->data(0).toString();

}

if (index.column() == 1) { // Colour Column

return QString("%1,%2,%3")

.arg(item->getColourR())

.arg(item->getColourG())

.arg(item->getColourB());

}

}

// Ensure visibility is correctly retrieved for checkboxes

if (role == Qt::CheckStateRole && index.column() == 2) {

return item->visible() ? Qt::Checked : Qt::Unchecked;

}

return QVariant();

}

Qt::ItemFlags ModelPartList::flags(const QModelIndex& index) const {

if (!index.isValid())

return Qt::NoItemFlags;

Qt::ItemFlags flags = QAbstractItemModel::flags(index);

if (index.column() == 2) { // Make the visibility column interactive

flags |= Qt::ItemIsUserCheckable | Qt::ItemIsEditable; // Enable checkbox interaction

}

return flags;

}

QVariant ModelPartList::headerData(int section, Qt::Orientation orientation, int role) const {

if (orientation == Qt::Horizontal && role == Qt::DisplayRole) {

switch (section) {

case 0: return "Part";

case 1: return "Colour";

case 2: return "Visible?";

}

}

return QVariant();

}

QModelIndex ModelPartList::index(int row, int column, const QModelIndex& parent) const {

ModelPart\* parentItem;

if( !parent.isValid() || !hasIndex(row, column, parent) )

parentItem = rootItem; // default to selecting root

else

parentItem = static\_cast<ModelPart\*>(parent.internalPointer());

ModelPart\* childItem = parentItem->child(row);

if( childItem )

return createIndex(row, column, childItem);

return QModelIndex();

}

QModelIndex ModelPartList::parent( const QModelIndex& index ) const {

if (!index.isValid())

return QModelIndex();

ModelPart\* childItem = static\_cast<ModelPart\*>(index.internalPointer());

ModelPart\* parentItem = childItem->parentItem();

if( parentItem == rootItem )

return QModelIndex();

return createIndex( parentItem->row(), 0, parentItem );

}

int ModelPartList::rowCount( const QModelIndex& parent ) const {

ModelPart\* parentItem;

if( parent.column() > 0 )

return 0;

if( !parent.isValid() )

parentItem = rootItem;

else

parentItem = static\_cast<ModelPart\*>(parent.internalPointer());

return parentItem->childCount();

}

ModelPart\* ModelPartList::getRootItem() {

return rootItem;

}

QModelIndex ModelPartList::appendChild(QModelIndex& parent, const QList<QVariant>& data) {

ModelPart\* parentPart;

if (parent.isValid())

parentPart = static\_cast<ModelPart\*>(parent.internalPointer());

else {

parentPart = rootItem;

parent = createIndex(0, 0, rootItem );

}

beginInsertRows( parent, rowCount(parent), rowCount(parent) );

ModelPart\* childPart = new ModelPart( data, parentPart );

parentPart->appendChild(childPart);

QModelIndex child = createIndex(0, 0, childPart);

endInsertRows();

emit layoutChanged();

return child;

}

ModelPartlist.h

/\*\* @file ModelPartList.h

\*

\* EEEE2076 - Software Engineering & VR Project

\*

\* Template for model part list that will be used to create the trewview.

\*

\* P Evans 2022

\*/

#ifndef VIEWER\_MODELPARTLIST\_H

#define VIEWER\_MODELPARTLIST\_H

#include "ModelPart.h"

#include <QAbstractItemModel>

#include <QModelIndex>

#include <QVariant>

#include <QString>

#include <QList>

class ModelPart;

class ModelPartList : public QAbstractItemModel {

Q\_OBJECT /\*\*< A special Qt tag used to indicate that this is a special Qt class that might require preprocessing before compiling. \*/

public:

/\*\* Constructor

\* Arguments are standard arguments for this type of class but are not used in this example.

\* @param data is not used

\* @param parent is used by the parent class constructor

\*/

ModelPartList( const QString& data, QObject\* parent = NULL );

/\*\* Destructor

\* Frees root item allocated in constructor

\*/

~ModelPartList();

/\*\* Return column count

\* @param parent is not used

\* @return number of columns in the tree view - "Part" and "Visible", i.e. 2 in this case

\*/

int columnCount( const QModelIndex& parent ) const;

/\*\* This returns the value of a particular row (i.e. the item index) and

\* columns (i.e. either the "Part" or "Visible" property).

\* It is used by QT internally - this is how Qt retrieves the text to display in the TreeView

\* @param index in a stucture Qt uses to specify the row and column it wants data for

\* @param role is how Qt specifies what it wants to do with the data

\* @return a QVariant which is a generic variable used to represent any Qt class type, in this case the QVariant will be a string

\*/

QVariant data( const QModelIndex& index, int role ) const;

bool setData(const QModelIndex &index, const QVariant &value, int role = Qt::EditRole) override;

/\*\* Standard function used by Qt internally.

\* @param index in a stucture Qt uses to specify the row and column it wants data for

\* @return a Qt item flags

\*/

Qt::ItemFlags flags( const QModelIndex& index ) const;

/\*\* Standard function used by Qt internally.

\*/

QVariant headerData( int section, Qt::Orientation orientation, int role ) const;

/\*\* Get a valid QModelIndex for a location in the tree (row is the row in the tree under "parent"

\* or under the root of the tree if parent isnt specified. Column is either 0 = "Part" or 1 = "Visible"

\* in this example

\* @param row is the item index

\* @param column is 0 or 1 - part name or visible stringstream

\* @param parent where the row is referenced from, usually the tree root

\* @return the QModelIndex structure

\*/

QModelIndex index( int row, int column, const QModelIndex& parent ) const;

/\*\* Take a QModelIndex for an item, get a QModel Index for its parent

\* @param index of item

\* @return index of parent

\*/

QModelIndex parent( const QModelIndex& index ) const;

/\*\* Get number of rows (items) under an item in tree

\* @param is the parent, all items under this will be counted

\* @return number of children

\*/

int rowCount( const QModelIndex& parent ) const;

/\*\* Get a pointer to the root item of the tree

\* @return the root item pointer

\*/

ModelPart\* getRootItem();

/\*\*

\*/

QModelIndex appendChild( QModelIndex& parent, const QList<QVariant>& data );

private:

ModelPart \*rootItem; /\*\*< This is a pointer to the item at the base of the tree \*/

};

#endif

Optiondialog.cpp

#include "optiondialog.h"

#include "ui\_optiondialog.h"

OptionDialog::OptionDialog(QWidget \*parent)

: QDialog(parent)

, ui(new Ui::OptionDialog)

{

ui->setupUi(this);

// Set the correct range for RGB spin boxes

ui->spinBox\_R->setRange(0, 255);

ui->spinBox\_G->setRange(0, 255);

ui->spinBox\_B->setRange(0, 255);

}

OptionDialog::~OptionDialog()

{

delete ui;

}

QString OptionDialog::getInputText() const

{

return ui->lineEdit->text();

}

void OptionDialog::setInputText(const QString &text) {

ui->lineEdit->setText(text);

}

bool OptionDialog::isItemVisible() const {

return ui->checkBoxVisible->isChecked();

}

void OptionDialog::setItemVisible(bool visible) {

ui->checkBoxVisible->setChecked(visible);

}

void OptionDialog::setRGB(int R, int G, int B) {

ui->spinBox\_R->setValue(R);

ui->spinBox\_G->setValue(G);

ui->spinBox\_B->setValue(B);

}

int OptionDialog::getR() {

return ui->spinBox\_R->value();

}

int OptionDialog::getG() {

return ui->spinBox\_G->value();

}

int OptionDialog::getB() {

return ui->spinBox\_B->value();

}

OptionDialog.h

#ifndef OPTIONDIALOG\_H

#define OPTIONDIALOG\_H

#include <QDialog>

namespace Ui {

class OptionDialog;

}

class OptionDialog : public QDialog

{

Q\_OBJECT

public:

explicit OptionDialog(QWidget \*parent = nullptr);

~OptionDialog();

QString getInputText() const;

public:

void setInputText(const QString &text);

public:

bool isItemVisible() const;

void setItemVisible(bool visible);

public:

void setRGB(int R, int G, int B);

int getR();

int getG();

int getB();

private:

Ui::OptionDialog \*ui;

};

#endif // OPTIONDIALOG\_H

CMakeList.txt

cmake\_minimum\_required(VERSION 3.16)

project(WS7Ex3 VERSION 0.1 LANGUAGES CXX)

set(CMAKE\_AUTOUIC ON)

set(CMAKE\_AUTOMOC ON)

set(CMAKE\_AUTORCC ON)

set(CMAKE\_CXX\_STANDARD 17)

set(CMAKE\_CXX\_STANDARD\_REQUIRED ON)

find\_package(QT NAMES Qt6 Qt5 REQUIRED COMPONENTS Widgets)

find\_package(Qt${QT\_VERSION\_MAJOR} REQUIRED COMPONENTS Widgets)

#set(VTK\_DIR "C:\Program Files (x86)\VTK\include\vtk-9.4")

find\_package(VTK REQUIRED)#for VTK

set(PROJECT\_SOURCES

main.cpp

mainwindow.cpp

mainwindow.h

mainwindow.ui

ModelPart.cpp

ModelPart.h

ModelPartList.cpp

ModelPartList.h

optiondialog.cpp

optiondialog.h

optiondialog.ui

icons.qrc#what will happen if I include qrc here and delet the if?

)

if(${QT\_VERSION\_MAJOR} GREATER\_EQUAL 6)

qt\_add\_executable(WS7Ex3

MANUAL\_FINALIZATION

${PROJECT\_SOURCES}

)

# Define target properties for Android with Qt 6 as:

# set\_property(TARGET WS7Ex3 APPEND PROPERTY QT\_ANDROID\_PACKAGE\_SOURCE\_DIR

# ${CMAKE\_CURRENT\_SOURCE\_DIR}/android)

# For more information, see https://doc.qt.io/qt-6/qt-add-executable.html#target-creation

else()

if(ANDROID)

add\_library(WS7Ex3 SHARED

${PROJECT\_SOURCES}

)

# Define properties for Android with Qt 5 after find\_package() calls as:

# set(ANDROID\_PACKAGE\_SOURCE\_DIR "${CMAKE\_CURRENT\_SOURCE\_DIR}/android")

else()

add\_executable(WS7Ex3

${PROJECT\_SOURCES}

)

endif()

endif()

target\_link\_libraries(WS7Ex3 PRIVATE Qt${QT\_VERSION\_MAJOR}::Widgets ${VTK\_LIBRARIES})#added

# Qt for iOS sets MACOSX\_BUNDLE\_GUI\_IDENTIFIER automatically since Qt 6.1.

# If you are developing for iOS or macOS you should consider setting an

# explicit, fixed bundle identifier manually though.

if(${QT\_VERSION} VERSION\_LESS 6.1.0)

set(BUNDLE\_ID\_OPTION MACOSX\_BUNDLE\_GUI\_IDENTIFIER com.example.WS7Ex3)

endif()

set\_target\_properties(WS7Ex3 PROPERTIES

${BUNDLE\_ID\_OPTION}

MACOSX\_BUNDLE\_BUNDLE\_VERSION ${PROJECT\_VERSION}

MACOSX\_BUNDLE\_SHORT\_VERSION\_STRING ${PROJECT\_VERSION\_MAJOR}.${PROJECT\_VERSION\_MINOR}

MACOSX\_BUNDLE TRUE

WIN32\_EXECUTABLE TRUE

)

include(GNUInstallDirs)

install(TARGETS WS7Ex3

BUNDLE DESTINATION .

LIBRARY DESTINATION ${CMAKE\_INSTALL\_LIBDIR}

RUNTIME DESTINATION ${CMAKE\_INSTALL\_BINDIR}

)

if(QT\_VERSION\_MAJOR EQUAL 6)

qt\_finalize\_executable(WS7Ex3)

endif()

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

################################### This needs adding #######################################

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Copy across Open VR bindings that map controllers

# The program will expect to find these in the build dir when it runs

add\_custom\_target( VRBindings )

add\_custom\_command( TARGET VRBindings PRE\_BUILD

COMMAND ${CMAKE\_COMMAND} -E

copy\_directory ${CMAKE\_SOURCE\_DIR}/vrbindings ${CMAKE\_BINARY\_DIR}/ )

#^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^