#include "mainwindow.h"

//for tests

#include <ChamferDistanceTransform.h>

#include <SaitoSquaredDistanceTransform.h>

//qXJ\_db

#include <xj2DLabel.h>

#include <xj2DViewportObject.h>

#include <xjCameraSensor.h>

#include <xjColorScalesManager.h>

#include <xjFacet.h>

#include <xjFileUtils.h>

#include <xjGBLSensor.h>

#include <xjImage.h>

#include <xjKdTree.h>

#include <xjPlane.h>

#include <xjProgressDialog.h>

#include <xjQuadric.h>

#include <xjSphere.h>

#include <xjCylinder.h>

#include <xjSubMesh.h>

//qXJ\_io

#include <xjShiftAndScaleCloudDlg.h>

#include <BinFilter.h>

#include <DepthMapFileFilter.h>

//QXJ\_glWindow

#include <xjGLWidget.h>

#include <xjRenderingTools.h>

//local includes

#include "xjConsole.h"

#include "xjEntityAction.h"

#include "xjHistogramWindow.h"

#include "xjInnerRect2DFinder.h"

//common

#include <xjPickingHub.h>

//common dialogs

#include <xjCameraParamEditDlg.h>

#include <xjDisplayOptionsDlg.h>

#include <xjPickOneElementDlg.h>

#include <xjStereoModeDlg.h>

//dialogs

#include "xjAboutDialog.h"

#include "xjAdjustZoomDlg.h"

#include "xjAlignDlg.h" //Aurelien BEY

#include "xjApplication.h"

#include "xjApplyTransformationDlg.h"

#include "xjAskThreeDoubleValuesDlg.h"

#include "xjBoundingBoxEditorDlg.h"

#include "xjCamSensorProjectionDlg.h"

#include "xjClippingBoxTool.h"

#include "xjColorFromScalarDlg.h"

#include "xjColorScaleEditorDlg.h"

#include "xjComparisonDlg.h"

#include "xjPrimitiveDistanceDlg.h"

#include "xjFilterByValueDlg.h"

#include "xjGBLSensorProjectionDlg.h"

#include "xjGeomFeaturesDlg.h"

#include "xjGraphicalSegmentationTool.h"

#include "xjGraphicalTransformationTool.h"

#include "xjItemSelectionDlg.h"

#include "xjLabelingDlg.h"

#include "xjMatchScalesDlg.h"

#include "xjNoiseFilterDlg.h"

#include "xjOrderChoiceDlg.h"

#include "xjPlaneEditDlg.h"

#include "xjPointListPickingDlg.h"

#include "xjPointPairRegistrationDlg.h"

#include "xjPointPropertiesDlg.h" //Aurelien BEY

#include "xjPrimitiveFactoryDlg.h"

#include "xjPtsSamplingDlg.h"

#include "xjRasterizeTool.h"

#include "xjRegistrationDlg.h" //Aurelien BEY

#include "xjRenderToFileDlg.h"

#include "xjScaleDlg.h"

#include "xjSectionExtractionTool.h"

#include "xjSensorComputeDistancesDlg.h"

#include "xjSensorComputeScatteringAnglesDlg.h"

#include "xjSORFilterDlg.h"

#include "xjSubsamplingDlg.h" //Aurelien BEY

#include "xjTracePolylineTool.h"

#include "xjTranslationManager.h"

#include "xjUnrollDlg.h"

#include "xjVolumeCalcTool.h"

#include "xjWaveformDialog.h"

#include "xjEntitySelectionDlg.h"

//other

#include "xjCropTool.h"

#include "xjGLPluginInterface.h"

#include "xjPersistentSettings.h"

#include "xjRecentFiles.h"

#include "xjRegistrationTools.h"

#include "xjUtils.h"

#include "db\_tree/xjDBRoot.h"

#include "pluginManager/xjPluginUIManager.h"

//Gamepads

#ifdef XJ\_GAMEPADS\_SUPPORT

#include "devices/gamepad/xjGamepadManager.h"

#endif

//Qt UI files

#include <ui\_distanceMapDlg.h>

#include <ui\_globalShiftSettingsDlg.h>

#include <ui\_mainWindow.h>

//System

#include <iostream>

#include <random>

//global static pointer (as there should only be one instance of MainWindow!)

static MainWindow\* s\_instance = nullptr;

//default file filter separator

static const QString s\_fileFilterSeparator(";;");

enum PickingOperation { NO\_PICKING\_OPERATION,

PICKING\_ROTATION\_CENTER,

PICKING\_LEVEL\_POINTS,

};

static xjGLWindow\* s\_pickingWindow = nullptr;

static PickingOperation s\_currentPickingOperation = NO\_PICKING\_OPERATION;

static std::vector<xj2DLabel\*> s\_levelLabels;

static xjPointCloud\* s\_levelMarkersCloud = nullptr;

static xjHObject\* s\_levelEntity = nullptr;

static QFileDialog::Options XJFileDialogOptions()

{

//dialog options

QFileDialog::Options dialogOptions = QFileDialog::Options();

if (!xjOptions::Instance().useNativeDialogs)

{

dialogOptions |= QFileDialog::DontUseNativeDialog;

}

return dialogOptions;

}

MainWindow::MainWindow()

: m\_UI( new Ui::MainWindow )

, m\_xjRoot(nullptr)

, m\_uiFrozen(false)

, m\_recentFiles(new xjRecentFiles(this))

, m\_3DMouseManager(nullptr)

, m\_gamepadManager(nullptr)

, m\_viewModePopupButton(nullptr)

, m\_pivotVisibilityPopupButton(nullptr)

, m\_FirstShow(true)

, m\_pickingHub(nullptr)

, m\_cpeDlg(nullptr)

, m\_gsTool(nullptr)

, m\_tplTool(nullptr)

, m\_seTool(nullptr)

, m\_transTool(nullptr)

, m\_clipTool(nullptr)

, m\_compDlg(nullptr)

, m\_ppDlg(nullptr)

, m\_plpDlg(nullptr)

, m\_pprDlg(nullptr)

, m\_pfDlg(nullptr)

{

m\_UI->setupUi( this );

setWindowTitle(tr("点云特征分析软件 v1.0"));

m\_pluginUIManager = new xjPluginUIManager( this, this );

xjTranslationManager::get().populateMenu( m\_UI->menuLanguage, xjApp->translationPath() );

// Set up dynamic menus

m\_UI->menuFile->insertMenu(m\_UI->actionSave, m\_recentFiles->menu());

//Console

xjConsole::Init(m\_UI->consoleWidget, this, this);

m\_UI->actionEnableQtWarnings->setChecked(xjConsole::QtMessagesEnabled());

//advanced widgets not handled by QDesigner

{

//view mode pop-up menu

{

m\_viewModePopupButton = new QToolButton();

QMenu\* menu = new QMenu(m\_viewModePopupButton);

menu->addAction(m\_UI->actionSetOrthoView);

menu->addAction(m\_UI->actionSetCenteredPerspectiveView);

menu->addAction(m\_UI->actionSetViewerPerspectiveView);

m\_viewModePopupButton->setMenu(menu);

m\_viewModePopupButton->setPopupMode(QToolButton::InstantPopup);

m\_viewModePopupButton->setToolTip("Set current view mode");

m\_viewModePopupButton->setStatusTip(m\_viewModePopupButton->toolTip());

m\_UI->toolBarView->insertWidget(m\_UI->actionZoomAndCenter, m\_viewModePopupButton);

m\_viewModePopupButton->setEnabled(false);

}

//pivot center pop-up menu

{

m\_pivotVisibilityPopupButton = new QToolButton();

QMenu\* menu = new QMenu(m\_pivotVisibilityPopupButton);

menu->addAction(m\_UI->actionSetPivotAlwaysOn);

menu->addAction(m\_UI->actionSetPivotRotationOnly);

menu->addAction(m\_UI->actionSetPivotOff);

m\_pivotVisibilityPopupButton->setMenu(menu);

m\_pivotVisibilityPopupButton->setPopupMode(QToolButton::InstantPopup);

m\_pivotVisibilityPopupButton->setToolTip("Set pivot visibility");

m\_pivotVisibilityPopupButton->setStatusTip(m\_pivotVisibilityPopupButton->toolTip());

m\_UI->toolBarView->insertWidget(m\_UI->actionZoomAndCenter,m\_pivotVisibilityPopupButton);

m\_pivotVisibilityPopupButton->setEnabled(false);

}

}

//tabifyDockWidget(DockableDBTree,DockableProperties);

//db-tree

{

m\_xjRoot = new xjDBRoot(m\_UI->dbTreeView, m\_UI->propertiesTreeView, this);

connect(m\_xjRoot, &xjDBRoot::selectionChanged, this, &MainWindow::updateUIWithSelection);

connect(m\_xjRoot, &xjDBRoot::dbIsEmpty, [&]() { updateUIWithSelection(); updateMenus(); }); //we don't call updateUI because there's no need to update the properties dialog

connect(m\_xjRoot, &xjDBRoot::dbIsNotEmptyAnymore, [&]() { updateUIWithSelection(); updateMenus(); }); //we don't call updateUI because there's no need to update the properties dialog

}

//MDI Area

{

m\_mdiArea = new QMdiArea(this);

setCentralWidget(m\_mdiArea);

connect(m\_mdiArea, &QMdiArea::subWindowActivated, this, &MainWindow::updateMenus);

connect(m\_mdiArea, &QMdiArea::subWindowActivated, this, &MainWindow::on3DViewActivated);

m\_mdiArea->installEventFilter(this);

}

//picking hub

{

m\_pickingHub = new xjPickingHub(this, this);

connect(m\_mdiArea, &QMdiArea::subWindowActivated, m\_pickingHub, &xjPickingHub::onActiveWindowChanged);

}

connectActions();

new3DView(true);

setupInputDevices();

freezeUI(false);

updateUI();

QMainWindow::statusBar()->showMessage(QString("准备就绪"));

}

MainWindow::~MainWindow()

{

destroyInputDevices();

cancelPreviousPickingOperation(false); //just in case

assert(m\_xjRoot && m\_mdiArea);

m\_xjRoot->disconnect();

m\_mdiArea->disconnect();

//we don't want any other dialog/function to use the following structures

xjDBRoot\* xjRoot = m\_xjRoot;

m\_xjRoot = nullptr;

//remove all entities from 3D views before quitting to avoid any side-effect

//(this won't be done automatically since we've just reset m\_xjRoot)

xjRoot->getRootEntity()->setDisplay\_recursive(nullptr);

for (int i = 0; i < getGLWindowCount(); ++i)

{

getGLWindow(i)->setSceneDB(nullptr);

}

m\_cpeDlg = nullptr;

m\_gsTool = nullptr;

m\_seTool = nullptr;

m\_transTool = nullptr;

m\_clipTool = nullptr;

m\_compDlg = nullptr;

m\_ppDlg = nullptr;

m\_plpDlg = nullptr;

m\_pprDlg = nullptr;

m\_pfDlg = nullptr;

//release all 'overlay' dialogs

while (!m\_mdiDialogs.empty())

{

xjMDIDialogs mdiDialog = m\_mdiDialogs.back();

m\_mdiDialogs.pop\_back();

mdiDialog.dialog->disconnect();

mdiDialog.dialog->stop(false);

mdiDialog.dialog->setParent(nullptr);

delete mdiDialog.dialog;

}

//m\_mdiDialogs.clear();

m\_mdiArea->closeAllSubWindows();

if (xjRoot)

{

delete xjRoot;

xjRoot = nullptr;

}

delete m\_UI;

m\_UI = nullptr;

xjConsole::ReleaseInstance(false); //if we flush the console, it will try to display the console window while we are destroying everything!

}

void MainWindow::initPlugins( )

{

m\_pluginUIManager->init();

// Set up dynamic tool bars

addToolBar( Qt::RightToolBarArea, m\_pluginUIManager->glFiltersToolbar() );

addToolBar( Qt::RightToolBarArea, m\_pluginUIManager->mainPluginToolbar() );

}

void MainWindow::doEnableQtWarnings(bool state)

{

xjConsole::EnableQtMessages(state);

}

void MainWindow::increasePointSize()

{

//active window?

xjGLWindow\* win = getActiveGLWindow();

if (win)

{

win->setPointSize(win->getViewportParameters().defaultPointSize + 1);

win->redraw();

}

}

void MainWindow::decreasePointSize()

{

//active window?

xjGLWindow\* win = getActiveGLWindow();

if (win)

{

win->setPointSize(win->getViewportParameters().defaultPointSize - 1);

win->redraw();

}

}

void MainWindow::connectActions()

{

assert(m\_xjRoot);

assert(m\_mdiArea);

//Keyboard shortcuts

//'A': toggles selected items activation

connect(m\_UI->actionToggleActivation, &QAction::triggered, this, [=]() {

toggleSelectedEntitiesProperty( xjEntityAction::TOGGLE\_PROPERTY::ACTIVE );

});

//'V': toggles selected items visibility

connect(m\_UI->actionToggleVisibility, &QAction::triggered, this, [=]() {

toggleSelectedEntitiesProperty( xjEntityAction::TOGGLE\_PROPERTY::VISIBLE );

});

//'N': toggles selected items normals visibility

connect(m\_UI->actionToggleNormals, &QAction::triggered, this, [=]() {

toggleSelectedEntitiesProperty( xjEntityAction::TOGGLE\_PROPERTY::NORMALS );

});

//'C': toggles selected items colors visibility

connect(m\_UI->actionToggleColors, &QAction::triggered, this, [=]() {

toggleSelectedEntitiesProperty( xjEntityAction::TOGGLE\_PROPERTY::COLOR );

});

//'S': toggles selected items SF visibility

connect(m\_UI->actionToggleSF, &QAction::triggered, this, [=]() {

toggleSelectedEntitiesProperty( xjEntityAction::TOGGLE\_PROPERTY::SCALAR\_FIELD );

});

//'D': toggles selected items '3D name' visibility

connect(m\_UI->actionToggleShowName, &QAction::triggered, this, [=]() {

toggleSelectedEntitiesProperty( xjEntityAction::TOGGLE\_PROPERTY::NAME );

});

//'M': toggles selected items materials/textures visibility

connect(m\_UI->actionToggleMaterials, &QAction::triggered, this, [=]() {

toggleSelectedEntitiesProperty( xjEntityAction::TOGGLE\_PROPERTY::MATERIAL );

});

/\*\*\* MAIN MENU \*\*\*/

//"File" menu

connect(m\_UI->actionOpen, &QAction::triggered, this, &MainWindow::doActionLoadFile);

connect(m\_UI->actionSave, &QAction::triggered, this, &MainWindow::doActionSaveFile);

connect(m\_UI->actionGlobalShiftSettings, &QAction::triggered, this, &MainWindow::doActionGlobalShiftSeetings);

connect(m\_UI->actionPrimitiveFactory, &QAction::triggered, this, &MainWindow::doShowPrimitiveFactory);

connect(m\_UI->actionCloseAll, &QAction::triggered, this, &MainWindow::closeAll);

connect(m\_UI->actionQuit, &QAction::triggered, this, &QWidget::close);

//"Look" menu

connect(m\_UI->actionSetUniqueColor, &QAction::triggered, this, &MainWindow::doActionSetUniqueColor);

connect(m\_UI->actionSetColorGradient, &QAction::triggered, this, &MainWindow::doActionSetColorGradient);

connect(m\_UI->actionChangeColorLevels, &QAction::triggered, this, &MainWindow::doActionChangeColorLevels);

connect(m\_UI->actionColorize, &QAction::triggered, this, &MainWindow::doActionColorize);

connect(m\_UI->actionRGBToGreyScale, &QAction::triggered, this, &MainWindow::doActionRGBToGreyScale);

connect(m\_UI->actionInterpolateColors, &QAction::triggered, this, &MainWindow::doActionInterpolateColors);

connect(m\_UI->actionEnhanceRGBWithIntensities, &QAction::triggered, this, &MainWindow::doActionEnhanceRGBWithIntensities);

connect(m\_UI->actionColorFromScalarField, &QAction::triggered, this, &MainWindow::doActionColorFromScalars);

connect(m\_UI->actionClearColor, &QAction::triggered, this, [=]() {

clearSelectedEntitiesProperty( xjEntityAction::CLEAR\_PROPERTY::COLORS );

});

//"Analysis" menu

connect(m\_UI->actionComputeNormals, &QAction::triggered, this, &MainWindow::doActionComputeNormals);

connect(m\_UI->actionInvertNormals, &QAction::triggered, this, &MainWindow::doActionInvertNormals);

connect(m\_UI->actionConvertNormalToHSV, &QAction::triggered, this, &MainWindow::doActionConvertNormalsToHSV);

connect(m\_UI->actionConvertNormalToDipDir, &QAction::triggered, this, &MainWindow::doActionConvertNormalsToDipDir);

connect(m\_UI->actionExportNormalToSF, &QAction::triggered, this, &MainWindow::doActionExportNormalToSF);

connect(m\_UI->actionOrientNormalsMST, &QAction::triggered, this, &MainWindow::doActionOrientNormalsMST);

connect(m\_UI->actionOrientNormalsFM, &QAction::triggered, this, &MainWindow::doActionOrientNormalsFM);

connect(m\_UI->actionClearNormals, &QAction::triggered, this, [=]() {

clearSelectedEntitiesProperty( xjEntityAction::CLEAR\_PROPERTY::NORMALS );

});

//"3D Views" menu

connect(m\_UI->menu3DViews, &QMenu::aboutToShow, this, &MainWindow::update3DViewsMenu);

connect(m\_UI->actionNew3DView, &QAction::triggered, this, &MainWindow::new3DView);

connect(m\_UI->actionZoomIn, &QAction::triggered, this, &MainWindow::zoomIn);

connect(m\_UI->actionZoomOut, &QAction::triggered, this, &MainWindow::zoomOut);

connect(m\_UI->actionClose3DView, &QAction::triggered, m\_mdiArea, &QMdiArea::closeActiveSubWindow);

connect(m\_UI->actionCloseAll3DViews, &QAction::triggered, m\_mdiArea, &QMdiArea::closeAllSubWindows);

connect(m\_UI->actionTile3DViews, &QAction::triggered, m\_mdiArea, &QMdiArea::tileSubWindows);

connect(m\_UI->actionCascade3DViews, &QAction::triggered, m\_mdiArea, &QMdiArea::cascadeSubWindows);

connect(m\_UI->actionNext3DView, &QAction::triggered, m\_mdiArea, &QMdiArea::activateNextSubWindow);

connect(m\_UI->actionPrevious3DView, &QAction::triggered, m\_mdiArea, &QMdiArea::activatePreviousSubWindow);

/\*\*\* Toolbars \*\*\*/

//View toolbar

connect(m\_UI->actionGlobalZoom, &QAction::triggered, this, &MainWindow::setGlobalZoom);

connect(m\_UI->actionPickRotationCenter, &QAction::triggered, this, &MainWindow::doPickRotationCenter);

connect(m\_UI->actionZoomAndCenter, &QAction::triggered, this, &MainWindow::zoomOnSelectedEntities);

connect(m\_UI->actionSetPivotAlwaysOn, &QAction::triggered, this, &MainWindow::setPivotAlwaysOn);

connect(m\_UI->actionSetPivotRotationOnly, &QAction::triggered, this, &MainWindow::setPivotRotationOnly);

connect(m\_UI->actionSetPivotOff, &QAction::triggered, this, &MainWindow::setPivotOff);

connect(m\_UI->actionSetOrthoView, &QAction::triggered, this, [this] () {

setOrthoView( getActiveGLWindow() );

});

connect(m\_UI->actionSetCenteredPerspectiveView, &QAction::triggered, this, [this] () {

setCenteredPerspectiveView( getActiveGLWindow() );

});

connect(m\_UI->actionSetViewerPerspectiveView, &QAction::triggered, this, [this] () {

setViewerPerspectiveView( getActiveGLWindow() );

});

connect(m\_UI->actionEnableStereo, &QAction::toggled, this, &MainWindow::toggleActiveWindowStereoVision);

connect(m\_UI->actionAutoPickRotationCenter, &QAction::toggled, this, &MainWindow::toggleActiveWindowAutoPickRotCenter);

connect(m\_UI->actionSetViewTop, &QAction::triggered, this, [=]() { setView( XJ\_TOP\_VIEW ); });

connect(m\_UI->actionSetViewBottom, &QAction::triggered, this, [=]() { setView( XJ\_BOTTOM\_VIEW ); });

connect(m\_UI->actionSetViewFront, &QAction::triggered, this, [=]() { setView( XJ\_FRONT\_VIEW ); });

connect(m\_UI->actionSetViewBack, &QAction::triggered, this, [=]() { setView( XJ\_BACK\_VIEW ); });

connect(m\_UI->actionSetViewLeft, &QAction::triggered, this, [=]() { setView( XJ\_LEFT\_VIEW ); });

connect(m\_UI->actionSetViewRight, &QAction::triggered, this, [=]() { setView( XJ\_RIGHT\_VIEW ); });

connect(m\_UI->actionSetViewIso1, &QAction::triggered, this, [=]() { setView( XJ\_ISO\_VIEW\_1 ); });

connect(m\_UI->actionSetViewIso2, &QAction::triggered, this, [=]() { setView( XJ\_ISO\_VIEW\_2 ); });

//hidden

connect(m\_UI->actionEnableVisualDebugTraces, &QAction::triggered, this, &MainWindow::toggleVisualDebugTraces);

}

typedef std::pair<xjHObject\*, xjGenericPointCloud\*> EntityCloudAssociation;

void MainWindow::doActionApplyScale()

{

xjScaleDlg dlg(this);

if (!dlg.exec())

return;

dlg.saveState();

//save values for next time

XJVector3d scales = dlg.getScales();

bool keepInPlace = dlg.keepInPlace();

bool rescaleGlobalShift = dlg.rescaleGlobalShift();

//we must backup 'm\_selectedEntities' as removeObjectTemporarilyFromDBTree can modify it!

xjHObject::Container selectedEntities = m\_selectedEntities;

//first check that all coordinates are kept 'small'

std::vector< EntityCloudAssociation > candidates;

{

bool testBigCoordinates = true;

//size\_t processNum = 0;

for (xjHObject \*entity : selectedEntities) //warning, getSelectedEntites may change during this loop!

{

bool lockedVertices;

//try to get the underlying cloud (or the vertices set for a mesh)

xjGenericPointCloud\* cloud = xjHObjectCaster::ToGenericPointCloud(entity, &lockedVertices);

//otherwise we can look if the selected entity is a polyline

if (!cloud && entity->isA(XJ\_TYPES::POLY\_LINE))

{

cloud = dynamic\_cast<xjGenericPointCloud\*>(static\_cast<xjPolyline\*>(entity)->getAssociatedCloud());

if (!cloud || cloud->isAncestorOf(entity))

lockedVertices = true;

}

if (!cloud || !cloud->isKindOf(XJ\_TYPES::POINT\_CLOUD))

{

xjLog::Warning(QString("[Apply scale] Entity '%1' can't be scaled this way").arg(entity->getName()));

continue;

}

if (lockedVertices)

{

xjUtils::DisplayLockedVerticesWarning(entity->getName(), haveOneSelection());

//++processNum;

continue;

}

XJVector3 C(0, 0, 0);

if (keepInPlace)

C = cloud->getOwnBB().getCenter();

//we must check that the resulting cloud coordinates are not too big

if (testBigCoordinates)

{

xjBBox bbox = cloud->getOwnBB();

XJVector3 bbMin = bbox.minCorner();

XJVector3 bbMax = bbox.maxCorner();

double maxx = std::max(std::abs(bbMin.x), std::abs(bbMax.x));

double maxy = std::max(std::abs(bbMin.y), std::abs(bbMax.y));

double maxz = std::max(std::abs(bbMin.z), std::abs(bbMax.z));

const double maxCoord = xjGlobalShiftManager::MaxCoordinateAbsValue();

bool oldCoordsWereTooBig = ( maxx > maxCoord

|| maxy > maxCoord

|| maxz > maxCoord );

if (!oldCoordsWereTooBig)

{

maxx = std::max(std::abs((bbMin.x - C.x) \* scales.x + C.x), std::abs((bbMax.x - C.x) \* scales.x + C.x));

maxy = std::max(std::abs((bbMin.y - C.y) \* scales.y + C.y), std::abs((bbMax.y - C.y) \* scales.y + C.y));

maxz = std::max(std::abs((bbMin.z - C.z) \* scales.z + C.z), std::abs((bbMax.z - C.z) \* scales.z + C.z));

bool newCoordsAreTooBig = ( maxx > maxCoord

|| maxy > maxCoord

|| maxz > maxCoord );

if (newCoordsAreTooBig)

{

if (QMessageBox::question(

this,

"Big coordinates",

"Resutling coordinates will be too big (original precision may be lost!). Proceed anyway?",

QMessageBox::Yes,

QMessageBox::No) == QMessageBox::Yes)

{

//ok, we won't test anymore and proceed

testBigCoordinates = false;

}

else

{

//we stop the process

return;

}

}

}

}

assert(cloud);

candidates.emplace\_back(entity, cloud);

}

}

if (candidates.empty())

{

xjConsole::Warning("[Apply scale] No eligible entities (point clouds or meshes) were selected!");

return;

}

//now do the real scaling work

{

for ( auto &candidate : candidates )

{

xjHObject\* ent = candidate.first;

xjGenericPointCloud\* cloud = candidate.second;

XJVector3 C(0, 0, 0);

if (keepInPlace)

{

C = cloud->getOwnBB().getCenter();

}

//we temporarily detach entity, as it may undergo

//"severe" modifications (octree deletion, etc.) --> see xjPointCloud::scale

xjHObjectContext objContext = removeObjectTemporarilyFromDBTree(cloud);

cloud->scale( static\_cast<PointCoordinateType>(scales.x),

static\_cast<PointCoordinateType>(scales.y),

static\_cast<PointCoordinateType>(scales.z),

C );

putObjectBackIntoDBTree(cloud, objContext);

cloud->prepareDisplayForRefresh\_recursive();

//don't forget the 'global shift'!

//DGM: but not the global scale!

if (rescaleGlobalShift)

{

const XJVector3d& shift = cloud->getGlobalShift();

cloud->setGlobalShift( XJVector3d( shift.x\*scales.x,

shift.y\*scales.y,

shift.z\*scales.z) );

}

ent->prepareDisplayForRefresh\_recursive();

}

}

//reselect previously selected entities!

if (m\_xjRoot)

m\_xjRoot->selectEntities(selectedEntities);

if (!keepInPlace)

zoomOnSelectedEntities();

refreshAll();

updateUI();

}

/\* Bounding box P.C.A. fit PCA \*/

void MainWindow::doComputeBestFitBB()

{

if (QMessageBox::warning( this,

"This method is for test purpose only",

"Cloud(s) are going to be rotated while still displayed in their previous position! Proceed?",

QMessageBox::Yes | QMessageBox::No,

QMessageBox::No ) != QMessageBox::Yes)

{

return;

}

//backup selected entities as removeObjectTemporarilyFromDBTree can modify them

xjHObject::Container selectedEntities = getSelectedEntities();

for (xjHObject \*entity : selectedEntities) //warning, getSelectedEntites may change during this loop!

{

xjGenericPointCloud\* cloud = xjHObjectCaster::ToGenericPointCloud(entity);

if (cloud && cloud->isA(XJ\_TYPES::POINT\_CLOUD)) // TODO

{

XJLib::Neighbourhood Yk(cloud);

XJLib::SquareMatrixd covMat = Yk.computeCovarianceMatrix();

if (covMat.isValid())

{

XJLib::SquareMatrixd eigVectors;

std::vector<double> eigValues;

if (Jacobi<double>::ComputeEigenValuesAndVectors(covMat, eigVectors, eigValues, true))

{

Jacobi<double>::SortEigenValuesAndVectors(eigVectors, eigValues);

/\* xj: eigenValues \*/

xjLog::Print("eigenValues: "+QString::number(eigValues[0]) + ", " + QString::number(eigValues[1]) + ", " + QString::number(eigValues[2]));

xjLog::Print("eigenVector:");

xjGLMatrix trans;

GLfloat\* rotMat = trans.data();

for (unsigned j = 0; j < 3; ++j)

{

double u[3];

Jacobi<double>::GetEigenVector(eigVectors, j, u);

XJVector3 v(static\_cast<PointCoordinateType>(u[0]),

static\_cast<PointCoordinateType>(u[1]),

static\_cast<PointCoordinateType>(u[2]));

v.normalize();

rotMat[j\*4] = static\_cast<float>(v.x);

rotMat[j\*4+1] = static\_cast<float>(v.y);

rotMat[j\*4+2] = static\_cast<float>(v.z);

/\* xj: eigenVector \*/

xjLog::Print(QString::number(j) + ": " + QString::number(v.x)+", "+ QString::number(v.y)+", "+QString::number(v.z));

}

const XJVector3\* G = Yk.getGravityCenter();

assert(G);

trans.shiftRotationCenter(\*G);

cloud->setGLTransformation(trans);

trans.invert();

//we temporarily detach entity, as it may undergo

//"severe" modifications (octree deletion, etc.) --> see xjPointCloud::applyRigidTransformation

xjHObjectContext objContext = removeObjectTemporarilyFromDBTree(cloud);

static\_cast<xjPointCloud\*>(cloud)->applyRigidTransformation(trans);

putObjectBackIntoDBTree(cloud,objContext);

entity->prepareDisplayForRefresh\_recursive();

/\* xj \*/

xjLog::Print("after invert:"); //full precision

xjLog::Print(trans.toString(12, ' ')); //full precision

}

}

}

}

refreshAll();

}

void MainWindow::doActionFlagMeshVertices()

{

bool errors = false;

bool suxjess = false;

for ( xjHObject \*entity : getSelectedEntities() )

{

if (entity->isKindOf(XJ\_TYPES::MESH))

{

xjGenericMesh\* mesh = xjHObjectCaster::ToGenericMesh(entity);

xjPointCloud\* vertices = xjHObjectCaster::ToPointCloud(mesh ? mesh->getAssociatedCloud() : nullptr);

if (mesh && vertices)

{

//prepare a new scalar field

int sfIdx = vertices->getScalarFieldIndexByName(XJ\_DEFAULT\_MESH\_VERT\_FLAGS\_SF\_NAME);

if (sfIdx < 0)

{

sfIdx = vertices->addScalarField(XJ\_DEFAULT\_MESH\_VERT\_FLAGS\_SF\_NAME);

if (sfIdx < 0)

{

xjConsole::Warning(QString("Not enough memory to flag the vertices of mesh '%1'!").arg(mesh->getName()));

errors = true;

continue;

}

}

XJLib::ScalarField\* flags = vertices->getScalarField(sfIdx);

XJLib::MeshSamplingTools::EdgeConnectivityStats stats;

if (XJLib::MeshSamplingTools::flagMeshVerticesByType(mesh,flags,&stats))

{

vertices->setCurrentDisplayedScalarField(sfIdx);

xjScalarField\* sf = vertices->getCurrentDisplayedScalarField();

if (sf)

{

sf->setColorScale(xjColorScalesManager::GetDefaultScale(xjColorScalesManager::VERTEX\_QUALITY));

//sf->setColorRampSteps(3); //ugly :(

}

vertices->showSF(true);

mesh->showSF(true);

mesh->prepareDisplayForRefresh\_recursive();

suxjess = true;

//display stats in the Console as well

xjConsole::Print(QString("[Mesh Quality] Mesh '%1' edges: %2 total (normal: %3 / on hole borders: %4 / non-manifold: %5)").arg(entity->getName()).arg(stats.edgesCount).arg(stats.edgesSharedByTwo).arg(stats.edgesNotShared).arg(stats.edgesSharedByMore));

}

else

{

vertices->deleteScalarField(sfIdx);

sfIdx = -1;

xjConsole::Warning(QString("Not enough memory to flag the vertices of mesh '%1'!").arg(mesh->getName()));

errors = true;

}

}

else

{

assert(false);

}

}

}

refreshAll();

updateUI();

if (suxjess)

{

//display reminder

forceConsoleDisplay();

xjConsole::Print(QString("[Mesh Quality] SF flags: %1 (NORMAL) / %2 (BORDER) / (%3) NON-MANIFOLD").arg(XJLib::MeshSamplingTools::VERTEX\_NORMAL).arg(XJLib::MeshSamplingTools::VERTEX\_BORDER).arg(XJLib::MeshSamplingTools::VERTEX\_NON\_MANIFOLD));

}

if (errors)

{

xjConsole::Error("Error(s) oxjurred! Check the console...");

}

}

void MainWindow::doActionMeasureMeshVolume()

{

for ( xjHObject \*entity : getSelectedEntities() )

{

if (entity->isKindOf(XJ\_TYPES::MESH))

{

xjMesh\* mesh = xjHObjectCaster::ToMesh(entity);

if (mesh)

{

//we compute the mesh volume

double V = XJLib::MeshSamplingTools::computeMeshVolume(mesh);

//we force the console to display itself

forceConsoleDisplay();

xjConsole::Print(QString("[Mesh Volume] Mesh '%1': V=%2 (cube units)").arg(entity->getName()).arg(V));

//check that the mesh is closed

XJLib::MeshSamplingTools::EdgeConnectivityStats stats;

if (XJLib::MeshSamplingTools::computeMeshEdgesConnectivity(mesh, stats))

{

if (stats.edgesNotShared != 0)

{

xjConsole::Warning(QString("[Mesh Volume] The above volume might be invalid (mesh has holes)"));

}

else if (stats.edgesSharedByMore != 0)

{

xjConsole::Warning(QString("[Mesh Volume] The above volume might be invalid (mesh has non-manifold edges)"));

}

}

else

{

xjConsole::Warning(QString("[Mesh Volume] The above volume might be invalid (not enough memory to check if the mesh is closed)"));

}

}

else

{

assert(false);

}

}

}

}

void MainWindow::doActionCheckPointsInsideFrustum()

{

//there should be only one camera sensor in the current selection!

if (!haveOneSelection() || !m\_selectedEntities[0]->isKindOf(XJ\_TYPES::CAMERA\_SENSOR))

{

xjConsole::Error("Select one and only one camera sensor!");

return;

}

xjCameraSensor\* sensor = xjHObjectCaster::ToCameraSensor(m\_selectedEntities[0]);

if (!sensor)

return;

//we need a cloud to filter!

xjHObject\* defaultCloud = sensor->getParent() && sensor->getParent()->isA(XJ\_TYPES::POINT\_CLOUD) ? sensor->getParent() : nullptr;

xjPointCloud\* pointCloud = askUserToSelectACloud(defaultCloud, "Select a cloud to filter:");

if (!pointCloud)

{

return;

}

//comupte/get the point cloud's octree

xjOctree::Shared octree = pointCloud->getOctree();

if (!octree)

{

octree = pointCloud->computeOctree();

if (!octree)

{

xjConsole::Error("Failed to compute the octree!");

return;

}

}

assert(octree);

// filter octree then project the points

std::vector<unsigned> inCameraFrustum;

if (!octree->intersectWithFrustum(sensor, inCameraFrustum))

{

xjConsole::Error("Failed to intersect sensor frustum with octree!");

}

else

{

// scalar field

const char sfName[] = "Frustum visibility";

int sfIdx = pointCloud->getScalarFieldIndexByName(sfName);

if (inCameraFrustum.empty())

{

xjConsole::Error("No point fell inside the frustum!");

if (sfIdx >= 0)

pointCloud->deleteScalarField(sfIdx);

}

else

{

if (sfIdx < 0)

sfIdx = pointCloud->addScalarField(sfName);

if (sfIdx < 0)

{

xjLog::Error("Failed to allocate memory for output scalar field!");

return;

}

XJLib::ScalarField\* sf = pointCloud->getScalarField(sfIdx);

assert(sf);

if (sf)

{

sf->fill(0);

const ScalarType c\_insideValue = static\_cast<ScalarType>(1);

for ( unsigned index : inCameraFrustum )

{

sf->setValue(index, c\_insideValue);

}

sf->computeMinAndMax();

pointCloud->setCurrentDisplayedScalarField(sfIdx);

pointCloud->showSF(true);

pointCloud->redrawDisplay();

}

}

}

updateUI();

}

void MainWindow::doActionShowDepthBuffer()

{

if (!haveSelection())

return;

for ( xjHObject \*entity : getSelectedEntities() )

{

if (entity->isKindOf(XJ\_TYPES::GBL\_SENSOR))

{

xjGBLSensor\* sensor = static\_cast<xjGBLSensor\*>(m\_selectedEntities[0]);

if (sensor->getDepthBuffer().zBuff.empty())

{

//look for depending cloud

xjGenericPointCloud\* cloud = xjHObjectCaster::ToGenericPointCloud(entity->getParent());

if (cloud)

{

//force depth buffer computation

int errorCode;

if (!sensor->computeDepthBuffer(cloud, errorCode))

{

xjConsole::Error(xjGBLSensor::GetErrorString(errorCode));

}

}

else

{

xjConsole::Error(QString("Internal error: sensor ('%1') parent is not a point cloud!").arg(sensor->getName()));

return;

}

}

xjRenderingTools::ShowDepthBuffer(sensor, this);

}

}

}

void MainWindow::doActionExportDepthBuffer()

{

if (!haveSelection())

return;

//persistent settings

QSettings settings;

settings.beginGroup(xjPS::SaveFile());

QString currentPath = settings.value(xjPS::CurrentPath(), xjFileUtils::defaultDocPath()).toString();

QString filename = QFileDialog::getSaveFileName(this,

"Select output file",

currentPath,

DepthMapFileFilter::GetFileFilter(),

nullptr,

XJFileDialogOptions()

);

if (filename.isEmpty())

{

//process cancelled by user

return;

}

//save last saving location

settings.setValue(xjPS::CurrentPath(),QFileInfo(filename).absolutePath());

settings.endGroup();

xjHObject\* toSave = nullptr;

bool multEntities = false;

if (haveOneSelection())

{

toSave = m\_selectedEntities.front();

}

else

{

toSave = new xjHObject("Temp Group");

for ( xjHObject \*entity : getSelectedEntities() )

{

toSave->addChild(entity,xjHObject::DP\_NONE);

}

multEntities = true;

}

DepthMapFileFilter::SaveParameters parameters;

{

parameters.alwaysDisplaySaveDialog = true;

}

XJ\_FILE\_ERROR result = DepthMapFileFilter().saveToFile(toSave, filename, parameters);

if (result != XJ\_FERR\_NO\_ERROR)

{

FileIOFilter::DisplayErrorMessage(result,"saving",filename);

}

else

{

xjLog::Print(QString("[I/O] File '%1' saved suxjessfully").arg(filename));

}

if (multEntities)

{

delete toSave;

toSave = nullptr;

}

}

void MainWindow::doActionComputePointsVisibility()

{

//there should be only one camera sensor in the current selection!

if (!haveOneSelection() || !m\_selectedEntities[0]->isKindOf(XJ\_TYPES::GBL\_SENSOR))

{

xjConsole::Error("Select one and only one GBL/TLS sensor!");

return;

}

xjGBLSensor\* sensor = xjHObjectCaster::ToGBLSensor(m\_selectedEntities[0]);

if (!sensor)

return;

//we need a cloud to filter!

xjHObject\* defaultCloud = sensor->getParent() && sensor->getParent()->isA(XJ\_TYPES::POINT\_CLOUD) ? sensor->getParent() : nullptr;

xjPointCloud\* pointCloud = askUserToSelectACloud(defaultCloud, "Select a cloud to filter:");

if (!pointCloud)

{

return;

}

if (sensor->getDepthBuffer().zBuff.empty())

{

if (defaultCloud)

{

//the sensor has no depth buffer, we'll ask the user if he wants to compute it first

if (QMessageBox::warning( this,

"Depth buffer.",

"Sensor has no depth buffer: do you want to compute it now?",

QMessageBox::Yes | QMessageBox::No,

QMessageBox::Yes ) == QMessageBox::No)

{

//we can stop then...

return;

}

int errorCode;

if (sensor->computeDepthBuffer(static\_cast<xjPointCloud\*>(defaultCloud), errorCode))

{

xjRenderingTools::ShowDepthBuffer(sensor, this);

}

else

{

xjConsole::Error(xjGBLSensor::GetErrorString(errorCode));

return;

}

}

else

{

xjConsole::Error("Sensor has no depth buffer (and no associated cloud?)");

return;

}

}

// scalar field

const char sfName[] = "Sensor visibility";

int sfIdx = pointCloud->getScalarFieldIndexByName(sfName);

if (sfIdx < 0)

sfIdx = pointCloud->addScalarField(sfName);

if (sfIdx < 0)

{

xjLog::Error("Failed to allocate memory for output scalar field!");

return;

}

XJLib::ScalarField\* sf = pointCloud->getScalarField(sfIdx);

assert(sf);

if (sf)

{

sf->fill(0);

//progress bar

xjProgressDialog pdlg(true);

XJLib::NormalizedProgress nprogress(&pdlg,pointCloud->size());

pdlg.setMethodTitle(tr("Compute visibility"));

pdlg.setInfo(tr("Points: %L1").arg( pointCloud->size() ));

pdlg.start();

QApplication::processEvents();

for (unsigned i = 0; i < pointCloud->size(); i++)

{

const XJVector3\* P = pointCloud->getPoint(i);

unsigned char visibility = sensor->checkVisibility(\*P);

ScalarType visValue = static\_cast<ScalarType>(visibility);

sf->setValue(i, visValue);

if (!nprogress.oneStep())

{

//cancelled by user

pointCloud->deleteScalarField(sfIdx);

sf = nullptr;

break;

}

}

if (sf)

{

sf->computeMinAndMax();

pointCloud->setCurrentDisplayedScalarField(sfIdx);

pointCloud->showSF(true);

xjConsole::Print(QString("Visibility computed for cloud '%1'").arg(pointCloud->getName()));

xjConsole::Print(QString("\tVisible = %1").arg(POINT\_VISIBLE));

xjConsole::Print(QString("\tHidden = %1").arg(POINT\_HIDDEN));

xjConsole::Print(QString("\tOut of range = %1").arg(POINT\_OUT\_OF\_RANGE));

xjConsole::Print(QString("\tOut of fov = %1").arg(POINT\_OUT\_OF\_FOV));

}

pointCloud->redrawDisplay();

}

updateUI();

}

void MainWindow::doActionConvertTextureToColor()

{

if ( !xjEntityAction::convertTextureToColor(m\_selectedEntities, this) )

return;

refreshAll();

updateUI();

}

void MainWindow::doActionSamplePointsOnMesh()

{

static unsigned s\_ptsSamplingCount = 1000000;

static double s\_ptsSamplingDensity = 10.0;

static bool s\_ptsSampleNormals = true;

static bool s\_useDensity = false;

xjPtsSamplingDlg dlg(this);

//restore last parameters

dlg.setPointsNumber(s\_ptsSamplingCount);

dlg.setDensityValue(s\_ptsSamplingDensity);

dlg.setGenerateNormals(s\_ptsSampleNormals);

dlg.setUseDensity(s\_useDensity);

if (!dlg.exec())

return;

xjProgressDialog pDlg(false, this);

pDlg.setAutoClose(false);

bool withNormals = dlg.generateNormals();

bool withRGB = dlg.interpolateRGB();

bool withTexture = dlg.interpolateTexture();

s\_useDensity = dlg.useDensity();

assert(dlg.getPointsNumber() >= 0);

s\_ptsSamplingCount = static\_cast<unsigned>(dlg.getPointsNumber());

s\_ptsSamplingDensity = dlg.getDensityValue();

s\_ptsSampleNormals = withNormals;

bool errors = false;

for ( xjHObject \*entity : getSelectedEntities() )

{

if (!entity->isKindOf(XJ\_TYPES::MESH))

continue;

xjGenericMesh\* mesh = xjHObjectCaster::ToGenericMesh(entity);

assert(mesh);

xjPointCloud\* cloud = mesh->samplePoints( s\_useDensity,

s\_useDensity ? s\_ptsSamplingDensity : s\_ptsSamplingCount,

withNormals,

withRGB,

withTexture,

&pDlg );

if (cloud)

{

addToDB(cloud);

}

else

{

errors = true;

}

}

if (errors)

xjLog::Error("[doActionSamplePointsOnMesh] Errors oxjurred during the process! Result may be incomplete!");

refreshAll();

}

void MainWindow::doActionSamplePointsOnPolyline()

{

static unsigned s\_ptsSamplingCount = 1000;

static double s\_ptsSamplingDensity = 10.0;

static bool s\_useDensity = false;

xjPtsSamplingDlg dlg(this);

//restore last parameters

dlg.setPointsNumber(s\_ptsSamplingCount);

dlg.setDensityValue(s\_ptsSamplingDensity);

dlg.setUseDensity(s\_useDensity);

dlg.optionsFrame->setVisible(false);

if (!dlg.exec())

return;

assert(dlg.getPointsNumber() >= 0);

s\_ptsSamplingCount = static\_cast<unsigned>(dlg.getPointsNumber());

s\_ptsSamplingDensity = dlg.getDensityValue();

s\_useDensity = dlg.useDensity();

bool errors = false;

for (xjHObject \*entity : getSelectedEntities())

{

if (!entity->isKindOf(XJ\_TYPES::POLY\_LINE))

continue;

xjPolyline\* poly = xjHObjectCaster::ToPolyline(entity);

assert(poly);

xjPointCloud\* cloud = poly->samplePoints( s\_useDensity,

s\_useDensity ? s\_ptsSamplingDensity : s\_ptsSamplingCount,

true);

if (cloud)

{

addToDB(cloud);

}

else

{

errors = true;

}

}

if (errors)

{

xjLog::Error("[doActionSamplePointsOnPolyline] Errors oxjurred during the process! Result may be incomplete!");

}

refreshAll();

}

void MainWindow::doRemoveDuplicatePoints()

{

if (!haveSelection())

return;

bool first = true;

//persistent setting(s)

QSettings settings;

settings.beginGroup(xjPS::DuplicatePointsGroup());

double minDistanceBetweenPoints = settings.value(xjPS::DuplicatePointsMinDist(),1.0e-12).toDouble();

bool ok;

minDistanceBetweenPoints = QInputDialog::getDouble(this, "Remove duplicate points", "Min distance between points:", minDistanceBetweenPoints, 0, 1.0e8, 12, &ok);

if (!ok)

return;

//save parameter

settings.setValue(xjPS::DuplicatePointsMinDist(), minDistanceBetweenPoints);

static const char DEFAULT\_DUPLICATE\_TEMP\_SF\_NAME[] = "DuplicateFlags";

xjProgressDialog pDlg(true, this);

pDlg.setAutoClose(false);

xjHObject::Container selectedEntities = getSelectedEntities(); //we have to use a local copy: 'unselectAllEntities' and 'selectEntity' will change the set of currently selected entities!

for (xjHObject \*entity : selectedEntities)

{

xjPointCloud\* cloud = xjHObjectCaster::ToPointCloud(entity);

if (cloud)

{

//create temporary SF for 'duplicate flags'

int sfIdx = cloud->getScalarFieldIndexByName(DEFAULT\_DUPLICATE\_TEMP\_SF\_NAME);

if (sfIdx < 0)

sfIdx = cloud->addScalarField(DEFAULT\_DUPLICATE\_TEMP\_SF\_NAME);

if (sfIdx >= 0)

cloud->setCurrentScalarField(sfIdx);

else

{

xjConsole::Error("Couldn't create temporary scalar field! Not enough memory?");

break;

}

xjOctree::Shared octree = cloud->getOctree();

XJLib::GeometricalAnalysisTools::ErrorCode result = XJLib::GeometricalAnalysisTools::FlagDuplicatePoints( cloud,

minDistanceBetweenPoints,

&pDlg,

octree.data());

if (result == XJLib::GeometricalAnalysisTools::NoError)

{

//count the number of duplicate points!

XJLib::ScalarField\* flagSF = cloud->getScalarField(sfIdx);

unsigned duplicateCount = 0;

assert(flagSF);

if (duplicateCount == 0)

{

xjConsole::Print(QString("Cloud '%1' has no duplicate points").arg(cloud->getName()));

}

else

{

xjConsole::Warning(QString("Cloud '%1' has %2 duplicate point(s)").arg(cloud->getName()).arg(duplicateCount));

xjPointCloud\* filteredCloud = cloud->filterPointsByScalarValue(0, 0);

if (filteredCloud)

{

int sfIdx2 = filteredCloud->getScalarFieldIndexByName(DEFAULT\_DUPLICATE\_TEMP\_SF\_NAME);

assert(sfIdx2 >= 0);

filteredCloud->deleteScalarField(sfIdx2);

filteredCloud->setName(QString("%1.clean").arg(cloud->getName()));

filteredCloud->setDisplay(cloud->getDisplay());

filteredCloud->prepareDisplayForRefresh();

if (toFilter.empty())

return;

double minVald = 0.0;

double maxVald = 1.0;

//compute min and max "displayed" scalar values of currently selected

//entities (the ones with an active scalar field only!)

{

for (size\_t i = 0; i < toFilter.size(); ++i)

{

xjScalarField\* sf = toFilter[i].second->getCurrentDisplayedScalarField();

assert(sf);

if (i == 0)

{

minVald = static\_cast<double>(sf->displayRange().start());

maxVald = static\_cast<double>(sf->displayRange().stop());

}

else

{

if (minVald > static\_cast<double>(sf->displayRange().start()))

minVald = static\_cast<double>(sf->displayRange().start());

if (maxVald < static\_cast<double>(sf->displayRange().stop()))

maxVald = static\_cast<double>(sf->displayRange().stop());

}

}

}

xjFilterByValueDlg dlg(minVald, maxVald, -1.0e9, 1.0e9, this);

if (!dlg.exec())

return;

xjFilterByValueDlg::Mode mode = dlg.mode();

assert(mode != xjFilterByValueDlg::CANCEL);

ScalarType minVal = static\_cast<ScalarType>(dlg.minDoubleSpinBox->value());

ScalarType maxVal = static\_cast<ScalarType>(dlg.maxDoubleSpinBox->value());

xjHObject::Container results;

{

for ( auto &item : toFilter )

{

xjHObject\* ent = item.first;

xjPointCloud\* pc = item.second;

//XJLib::ScalarField\* sf = pc->getCurrentDisplayedScalarField();

//assert(sf);

//we set as output (OUT) the currently displayed scalar field

int outSfIdx = pc->getCurrentDisplayedScalarFieldIndex();

assert(outSfIdx >= 0);

pc->setCurrentOutScalarField(outSfIdx);

//pc->setCurrentScalarField(outSfIdx);

xjHObject\* resultInside = nullptr;

xjHObject\* resultOutside = nullptr;

if (ent->isKindOf(XJ\_TYPES::MESH))

{

pc->hidePointsByScalarValue(minVal, maxVal);

if (ent->isA(XJ\_TYPES::MESH)/\*|| ent->isKindOf(XJ\_TYPES::PRIMITIVE)\*/) //TODO

resultInside = xjHObjectCaster::ToMesh(ent)->createNewMeshFromSelection(false);

else if (ent->isA(XJ\_TYPES::SUB\_MESH))

resultInside = xjHObjectCaster::ToSubMesh(ent)->createNewSubMeshFromSelection(false);

if (mode == xjFilterByValueDlg::SPLIT)

{

pc->invertVisibilityArray();

if (ent->isA(XJ\_TYPES::MESH)/\*|| ent->isKindOf(XJ\_TYPES::PRIMITIVE)\*/) //TODO

resultOutside = xjHObjectCaster::ToMesh(ent)->createNewMeshFromSelection(false);

else if (ent->isA(XJ\_TYPES::SUB\_MESH))

resultOutside = xjHObjectCaster::ToSubMesh(ent)->createNewSubMeshFromSelection(false);

}

pc->unallocateVisibilityArray();

}

else if (ent->isKindOf(XJ\_TYPES::POINT\_CLOUD))

{

meshes.push\_back(mesh);

}

else

{

xjConsole::Warning(QString("Only meshes with standard vertices are handled for now! Can't merge entity '%1'...").arg(entity->getName()));

}

}

else

{

xjConsole::Warning(QString("Entity '%1' is neither a cloud nor a mesh, can't merge it!").arg(entity->getName()));

}

}

}

catch (const std::bad\_alloc&)

{

xjLog::Error("Not enough memory!");

return;

}

if (clouds.empty() && meshes.empty())

{

xjLog::Error("Select only clouds or meshes!");

return;

}

if (!clouds.empty() && !meshes.empty())

{

xjLog::Error("Can't mix point clouds and meshes!");

}

//merge clouds?

if (!clouds.empty())

{

//we deselect all selected entities (as most of them are going to disappear)

if (m\_xjRoot)

{

m\_xjRoot->unselectAllEntities();

assert(!haveSelection());

//m\_selectedEntities.clear();

}

//we will remove the useless clouds/meshes later

xjHObject::Container toBeRemoved;

xjPointCloud\* firstCloud = nullptr;

xjHObjectContext firstCloudContext;

//whether to generate the 'original cloud index' scalar field or not

XJLib::ScalarField\* ocIndexSF = nullptr;

size\_t cloudIndex = 0;

for (size\_t i = 0; i < clouds.size(); ++i)

{

xjPointCloud\* pc = clouds[i];

if (!firstCloud)

{

//we don't delete the first cloud (we'll merge the other one 'inside' it

firstCloud = pc;

//we still have to temporarily detach the first cloud, as it may undergo

//"severe" modifications (octree deletion, etc.) --> see xjPointCloud::operator +=

firstCloudContext = removeObjectTemporarilyFromDBTree(firstCloud);

if (QMessageBox::question(this, "Original cloud index", "Do you want to generate a scalar field with the original cloud index?") == QMessageBox::Yes)

{

int sfIdx = pc->getScalarFieldIndexByName(XJ\_ORIGINAL\_CLOUD\_INDEX\_SF\_NAME);

if (sfIdx < 0)

{

sfIdx = pc->addScalarField(XJ\_ORIGINAL\_CLOUD\_INDEX\_SF\_NAME);

}

if (sfIdx < 0)

{

xjConsole::Error("Couldn't allocate a new scalar field for storing the original cloud index! Try to free some memory ...");

QFile csvFile(outputFilename);

if (!csvFile.open(QFile::WriteOnly | QFile::Text))

{

xjConsole::Error("Failed to open file for writing! (check file permissions)");

return;

}

//save last saving location

settings.setValue(xjPS::CurrentPath(), QFileInfo(outputFilename).absolutePath());

settings.endGroup();

//write CSV header

QTextStream csvStream(&csvFile);

csvStream << "Name;";

csvStream << "Width;";

csvStream << "Height;";

csvStream << "Cx;";

csvStream << "Cy;";

csvStream << "Cz;";

csvStream << "Nx;";

csvStream << "Ny;";

csvStream << "Nz;";

csvStream << "Dip;";

csvStream << "Dip dir;";

csvStream << endl;

QChar separator(';');

//write one line per plane

for (xjHObject\* ent : planes)

{

xjPlane\* plane = static\_cast<xjPlane\*>(ent);

XJVector3 C = plane->getOwnBB().getCenter();

XJVector3 N = plane->getNormal();

PointCoordinateType dip\_deg = 0;

PointCoordinateType dipDir\_deg = 0;

xjNormalVectors::ConvertNormalToDipAndDipDir(N, dip\_deg, dipDir\_deg);

csvStream << plane->getName() << separator; //Name

csvStream << plane->getXWidth() << separator; //Width

csvStream << plane->getYWidth() << separator; //Height

csvStream << C.x << separator; //Cx

csvStream << C.y << separator; //Cy

csvStream << C.z << separator; //Cz

csvStream << N.x << separator; //Nx

csvStream << N.y << separator; //Ny

csvStream << N.z << separator; //Nz

csvStream << dip\_deg << separator; //Dip

csvStream << dipDir\_deg << separator; //Dip direction

csvStream << endl;

}

xjConsole::Print(QString("[I/O] File '%1' suxjessfully saved (%2 plane(s))").arg(outputFilename).arg(planes.size()));

csvFile.close();

}

void MainWindow::doActionExportCloudInfo()

{

//look for clouds

xjHObject::Container clouds;

const xjHObject::Container& selectedEntities = getSelectedEntities();

if (selectedEntities.size() == 1 && selectedEntities.front()->isA(XJ\_TYPES::HIERARCHY\_OBJECT))

{

//a group

selectedEntities.front()->filterChildren(clouds, true, XJ\_TYPES::POINT\_CLOUD, true);

}

else

{

for (xjHObject\* entity : selectedEntities)

{

xjPointCloud\* cloud = xjHObjectCaster::ToPointCloud(entity);

if (cloud)

{

clouds.push\_back(cloud);

}

}

}

if (clouds.empty())

{

xjConsole::Error("Select at least one point cloud!");

return;

}

//persistent settings

QSettings settings;

settings.beginGroup(xjPS::SaveFile());

QString currentPath = settings.value(xjPS::CurrentPath(), xjFileUtils::defaultDocPath()).toString();

QString outputFilename = QFileDialog::getSaveFileName( this,

"Select output file",

currentPath,

"\*.csv",

nullptr,

XJFileDialogOptions());

if (outputFilename.isEmpty())

{

//process cancelled by the user

return;

}

QFile csvFile(outputFilename);

if (!csvFile.open(QFile::WriteOnly | QFile::Text))

{

xjConsole::Error("Failed to open file for writing! (check file permissions)");

return;

}

//save last saving location

settings.setValue(xjPS::CurrentPath(), QFileInfo(outputFilename).absolutePath());

settings.endGroup();

//determine the maximum number of SFs

unsigned maxSFCount = 0;

for (xjHObject\* entity : clouds)

{

maxSFCount = std::max<unsigned>(maxSFCount, static\_cast<xjPointCloud\*>(entity)->getNumberOfScalarFields());

}

//write CSV header

QTextStream csvStream(&csvFile);

csvStream << "Name;";

csvStream << "Points;";

csvStream << "meanX;";

csvStream << "meanY;";

csvStream << "meanZ;";

{

for (unsigned i = 0; i < maxSFCount; ++i)

{

QString sfIndex = QString("SF#%1").arg(i + 1);

csvStream << sfIndex << " name;";

csvStream << sfIndex << " valid values;";

csvStream << sfIndex << " mean;";

csvStream << sfIndex << " std.dev.;";

csvStream << sfIndex << " sum;";

}

}

csvStream << endl;

//write one line per cloud

{

for (xjHObject\* entity : clouds)

{

xjPointCloud\* cloud = static\_cast<xjPointCloud\*>(entity);

XJVector3 G = \*XJLib::Neighbourhood(cloud).getGravityCenter();

csvStream << cloud->getName() << ";" /\*"Name;"\*/;

csvStream << cloud->size() << ";" /\*"Points;"\*/;

csvStream << G.x << ";" /\*"meanX;"\*/;

csvStream << G.y << ";" /\*"meanY;"\*/;

csvStream << G.z << ";" /\*"meanZ;"\*/;

for (unsigned j = 0; j < cloud->getNumberOfScalarFields(); ++j)

{

XJLib::ScalarField\* sf = cloud->getScalarField(j);

csvStream << sf->getName() << ";" /\*"SF name;"\*/;

unsigned validCount = 0;

double sfSum = 0.0;

double sfSum2 = 0.0;

for (unsigned k = 0; k < sf->currentSize(); ++k)

{

const ScalarType& val = sf->getValue(k);

if (XJLib::ScalarField::ValidValue(val))

{

++validCount;

sfSum += val;

sfSum2 += val\*val;

}

}

csvStream << validCount << ";" /\*"SF valid values;"\*/;

double mean = sfSum/validCount;

csvStream << mean << ";" /\*"SF mean;"\*/;

csvStream << sqrt(std::abs(sfSum2/validCount - mean\*mean)) << ";" /\*"SF std.dev.;"\*/;

csvStream << sfSum << ";" /\*"SF sum;"\*/;

}

csvStream << endl;

}

}

xjConsole::Print(QString("[I/O] File '%1' suxjessfully saved (%2 cloud(s))").arg(outputFilename).arg(clouds.size()));

csvFile.close();

}

bool MainWindow::checkStereoMode(xjGLWindow\* win)

{

assert(win);

if (win && win->getViewportParameters().perspectiveView && win->stereoModeIsEnabled())

{

xjGLWindow::StereoParams params = win->getStereoParams();

bool wasExclusiveFullScreen = win->exclusiveFullScreen();

if (wasExclusiveFullScreen)

{

win->toggleExclusiveFullScreen(false);

}

win->disableStereoMode();

if (QMessageBox::question( this,

"Stereo mode",

"Stereo-mode only works in perspective mode. Do you want to disable it?",

QMessageBox::Yes,

QMessageBox::No) == QMessageBox::No )

{

if (wasExclusiveFullScreen)

{

win->toggleExclusiveFullScreen(true);

win->enableStereoMode(params);

}

return false;

}

else

{

if (win == getActiveGLWindow())

{

m\_UI->actionEnableStereo->setChecked(false);

}

else

{

assert(false);

m\_UI->actionEnableStereo->blockSignals(true);

m\_UI->actionEnableStereo->setChecked(false);

m\_UI->actionEnableStereo->blockSignals(false);

}

}

}

return true;

}

void MainWindow::toggleActiveWindowCenteredPerspective()

{

xjGLWindow\* win = getActiveGLWindow();

if (win)

{

const xjViewportParameters& params = win->getViewportParameters();

if (params.perspectiveView && params.objectCenteredView && !checkStereoMode(win)) //we need to check this only if we are already in object-centered perspective mode

{

return;

}

win->togglePerspective(true);

win->redraw(false);

updateViewModePopUpMenu(win);

updatePivotVisibilityPopUpMenu(win);

}

}

void MainWindow::toggleActiveWindowViewerBasedPerspective()

{

xjGLWindow\* win = getActiveGLWindow();

if (win)

{

const xjViewportParameters& params = win->getViewportParameters();

if (params.perspectiveView && !params.objectCenteredView && !checkStereoMode(win)) //we need to check this only if we are already in viewer-based perspective mode

{

return;

}

win->togglePerspective(false);

win->redraw(false);

updateViewModePopUpMenu(win);

updatePivotVisibilityPopUpMenu(win);

}

}

void MainWindow::toggleLockRotationAxis()

{

xjGLWindow\* win = getActiveGLWindow();

if (win)

{

bool wasLocked = win->isRotationAxisLocked();

bool isLocked = !wasLocked;

static XJVector3d s\_lastAxis(0.0, 0.0, 1.0);

if (isLocked)

{

xjAskThreeDoubleValuesDlg axisDlg("x", "y", "z", -1.0e12, 1.0e12, s\_lastAxis.x, s\_lastAxis.y, s\_lastAxis.z, 4, "Lock rotation axis", this);

if (axisDlg.buttonBox->button(QDialogButtonBox::Ok))

axisDlg.buttonBox->button(QDialogButtonBox::Ok)->setFocus();

if (!axisDlg.exec())

return;

s\_lastAxis.x = axisDlg.doubleSpinBox1->value();

s\_lastAxis.y = axisDlg.doubleSpinBox2->value();

s\_lastAxis.z = axisDlg.doubleSpinBox3->value();

}

win->lockRotationAxis(isLocked, s\_lastAxis);

m\_UI->actionLockRotationAxis->blockSignals(true);

m\_UI->actionLockRotationAxis->setChecked(isLocked);

m\_UI->actionLockRotationAxis->blockSignals(false);

if (isLocked)

{

win->displayNewMessage(QString("[ROTATION LOCKED]"), xjGLWindow::UPPER\_CENTER\_MESSAGE, false, 24 \* 3600, xjGLWindow::ROTAION\_LOCK\_MESSAGE);

}

else

{

win->displayNewMessage(QString(), xjGLWindow::UPPER\_CENTER\_MESSAGE, false, 0, xjGLWindow::ROTAION\_LOCK\_MESSAGE);

}

win->redraw(true, false);

}

}

void MainWindow::doActionEnableBubbleViewMode()

{

//special case: the selected entity is a TLS sensor or a cloud with a TLS sensor

if (m\_xjRoot)

{

xjHObject::Container selectedEntities;

m\_xjRoot->getSelectedEntities(selectedEntities);

if (selectedEntities.size() == 1)

{

xjHObject\* ent = selectedEntities.front();

xjGBLSensor\* sensor = nullptr;

if (ent->isA(XJ\_TYPES::GBL\_SENSOR))

{

sensor = static\_cast<xjGBLSensor\*>(ent);

}

else if (ent->isA(XJ\_TYPES::POINT\_CLOUD))

{

xjHObject::Container sensors;

ent->filterChildren(sensors, false, XJ\_TYPES::GBL\_SENSOR, true);

if (sensors.size() >= 1)

{

sensor = static\_cast<xjGBLSensor\*>(sensors.front());

}

}

if (sensor)

{

sensor->applyViewport();

return;

}

}

}

//otherwise we simply enable the bubble view mode in the active 3D view

xjGLWindow\* win = getActiveGLWindow();

if (win)

{

win->setBubbleViewMode(true);

win->redraw(false);

}

}

void MainWindow::removeFromDB(xjHObject\* obj, bool autoDelete/\*=true\*/)

{

if (!obj)

return;

//remove dependency to avoid deleting the object when removing it from DB tree

if (!autoDelete && obj->getParent())

obj->getParent()->removeDependencyWith(obj);

if (m\_xjRoot)

m\_xjRoot->removeElement(obj);

}

void MainWindow::setSelectedInDB(xjHObject\* obj, bool selected)

{

if (obj && m\_xjRoot)

{

if (selected)

m\_xjRoot->selectEntity(obj);

else

m\_xjRoot->unselectEntity(obj);

}

}

void MainWindow::addToDB( xjHObject\* obj,

bool updateZoom/\*=true\*/,

bool autoExpandDBTree/\*=true\*/,

bool checkDimensions/\*=true\*/,

bool autoRedraw/\*=true\*/)

{

//let's check that the new entity is not too big nor too far from scene center!

if (checkDimensions)

{

//get entity bounding box

xjBBox bBox = obj->getBB\_recursive();

XJVector3 center = bBox.getCenter();

PointCoordinateType diag = bBox.getDiagNorm();

XJVector3d P = XJVector3d::fromArray(center.u);

XJVector3d Pshift(0, 0, 0);

double scale = 1.0;

bool preserveCoordinateShift = true;

//here we must test that coordinates are not too big whatever the case because OpenGL

//really doesn't like big ones (even if we work with GLdoubles :( ).

if (xjGlobalShiftManager::Handle(P, diag, xjGlobalShiftManager::DIALOG\_IF\_NECESSARY, false, Pshift, &preserveCoordinateShift, &scale))

{

bool needRescale = (scale != 1.0);

bool needShift = (Pshift.norm2() > 0);

if (needRescale || needShift)

{

xjGLMatrix mat;

mat.toIdentity();

mat.data()[0] = mat.data()[5] = mat.data()[10] = static\_cast<float>(scale);

mat.setTranslation(Pshift);

obj->applyGLTransformation\_recursive(&mat);

xjConsole::Warning(QString("Entity '%1' has been translated: (%2,%3,%4) and rescaled of a factor %5 [original position will be restored when saving]").arg(obj->getName()).arg(Pshift.x,0,'f',2).arg(Pshift.y,0,'f',2).arg(Pshift.z,0,'f',2).arg(scale,0,'f',6));

}

//update 'global shift' and 'global scale' for ALL clouds recursively

if (preserveCoordinateShift)

{

//FIXME: why don't we do that all the time by the way?!

xjHObject::Container children;

children.push\_back(obj);

while (!children.empty())

{

xjHObject\* child = children.back();

children.pop\_back();

if (child->isKindOf(XJ\_TYPES::POINT\_CLOUD))

{

xjGenericPointCloud\* pc = xjHObjectCaster::ToGenericPointCloud(child);

pc->setGlobalShift(pc->getGlobalShift() + Pshift);

pc->setGlobalScale(pc->getGlobalScale() \* scale);

}

for (unsigned i = 0; i < child->getChildrenNumber(); ++i)

{

children.push\_back(child->getChild(i));

}

}

}

}

}

//add object to DB root

if (m\_xjRoot)

{

//force a 'global zoom' if the DB was emtpy!

if (!m\_xjRoot->getRootEntity() || m\_xjRoot->getRootEntity()->getChildrenNumber() == 0)

{

updateZoom = true;

}

m\_xjRoot->addElement(obj, autoExpandDBTree);

}

else

{

xjLog::Warning("[MainWindow::addToDB] Internal error: no associated db?!");

assert(false);

}

//we can now set destination display (if none already)

if (!obj->getDisplay())

{

xjGLWindow\* activeWin = getActiveGLWindow();

if (!activeWin)

{

//no active GL window?!

return;

}

obj->setDisplay\_recursive(activeWin);

}

//eventually we update the corresponding display

assert(obj->getDisplay());

if (updateZoom)

{

static\_cast<xjGLWindow\*>(obj->getDisplay())->zoomGlobal(); //automatically calls xjGLWindow::redraw

}

else if (autoRedraw)

{

obj->redrawDisplay();

}

}

void MainWindow::onExclusiveFullScreenToggled(bool state)

{

//we simply update the fullscreen action method icon (whatever the window)

xjGLWindow\* win = getActiveGLWindow();

if ( win == nullptr )

return;

m\_UI->actionExclusiveFullScreen->blockSignals(true);

m\_UI->actionExclusiveFullScreen->setChecked(win ? win->exclusiveFullScreen() : false);

m\_UI->actionExclusiveFullScreen->blockSignals(false);

if ( !state

&& win->stereoModeIsEnabled()

&& ( win->getStereoParams().glassType == xjGLWindow::StereoParams::NVIDIA\_VISION

|| win->getStereoParams().glassType == xjGLWindow::StereoParams::GENERIC\_STEREO\_DISPLAY ))

{

//auto disable stereo mode as NVidia Vision only works in full screen mode!

m\_UI->actionEnableStereo->setChecked(false);

}

}

void MainWindow::addToDBAuto(const QStringList& filenames)

{

xjGLWindow\* win = qobject\_cast<xjGLWindow\*>(QObject::sender());

addToDB(filenames, QString(), win);

}

void MainWindow::addToDB( const QStringList& filenames,

QString fileFilter/\*=QString()\*/,

xjGLWindow\* destWin/\*=0\*/)

{

//to use the same 'global shift' for multiple files

XJVector3d loadCoordinatesShift(0,0,0);

bool loadCoordinatesTransEnabled = false;

FileIOFilter::LoadParameters parameters;

{

parameters.alwaysDisplayLoadDialog = true;

parameters.shiftHandlingMode = xjGlobalShiftManager::DIALOG\_IF\_NECESSARY;

parameters.coordinatesShift = &loadCoordinatesShift;

parameters.coordinatesShiftEnabled = &loadCoordinatesTransEnabled;

parameters.parentWidget = this;

}

//the same for 'addToDB' (if the first one is not supported, or if the scale remains too big)

XJVector3d addCoordinatesShift(0, 0, 0);

const xjOptions& options = xjOptions::Instance();

FileIOFilter::ResetSesionCounter();

for ( const QString &filename : filenames )

{

XJ\_FILE\_ERROR result = XJ\_FERR\_NO\_ERROR;

xjHObject\* newGroup = FileIOFilter::LoadFromFile(filename, parameters, result, fileFilter);

if (newGroup)

{

if (!options.normalsDisplayedByDefault)

{

//disable the normals on all loaded clouds!

xjHObject::Container clouds;

newGroup->filterChildren(clouds, true, XJ\_TYPES::POINT\_CLOUD);

for (xjHObject\* cloud : clouds)

{

if (cloud)

{

static\_cast<xjGenericPointCloud\*>(cloud)->showNormals(false);

}

}

}

if (destWin)

{

newGroup->setDisplay\_recursive(destWin);

}

addToDB(newGroup, true, true, false);

m\_recentFiles->addFilePath( filename );

}

if (result == XJ\_FERR\_CANCELED\_BY\_USER)

{

//stop importing the file if the user has cancelled the current process!

break;

}

}

QMainWindow::statusBar()->showMessage(QString("%1 file(s) loaded").arg(filenames.size()),2000);

}

void MainWindow::handleNewLabel(xjHObject\* entity)

{

if (entity)

{

addToDB(entity);

}

else

{

assert(false);

}

}

void MainWindow::forceConsoleDisplay()

{

//if the console is hidden, we autoamtically display it!

if (m\_UI->DockableConsole && m\_UI->DockableConsole->isHidden())

{

m\_UI->DockableConsole->show();

QApplication::processEvents();

}

}

xjColorScalesManager\* MainWindow::getColorScalesManager()

{

return xjColorScalesManager::GetUniqueInstance();

}

void MainWindow::doActionLoadFile()

{

//persistent settings

QSettings settings;

settings.beginGroup(xjPS::LoadFile());

QString currentPath = settings.value(xjPS::CurrentPath(), xjFileUtils::defaultDocPath()).toString();

QString currentOpenDlgFilter = settings.value(xjPS::SelectedInputFilter(), BinFilter::GetFileFilter()).toString();

// Add all available file I/O filters (with import capabilities)

const QStringList filterStrings = FileIOFilter::ImportFilterList();

const QString &allFilter = filterStrings.at( 0 );

if ( !filterStrings.contains( currentOpenDlgFilter ) )

{

currentOpenDlgFilter = allFilter;

}

//file choosing dialog

QStringList selectedFiles = QFileDialog::getOpenFileNames( this,

tr("Open file(s)"),

currentPath,

filterStrings.join(s\_fileFilterSeparator),

&currentOpenDlgFilter,

XJFileDialogOptions());

if (selectedFiles.isEmpty())

return;

//save last loading parameters

currentPath = QFileInfo(selectedFiles[0]).absolutePath();

settings.setValue(xjPS::CurrentPath(),currentPath);

settings.setValue(xjPS::SelectedInputFilter(),currentOpenDlgFilter);

settings.endGroup();

if (currentOpenDlgFilter == allFilter)

{

currentOpenDlgFilter.clear(); //this way FileIOFilter will try to guess the file type automatically!

}

//load files

addToDB(selectedFiles, currentOpenDlgFilter);

}

//Helper: check for a filename validity

static bool IsValidFileName(QString filename)

{

#ifdef XJ\_WINDOWS

QString sPattern("^(?!^(PRN|AUX|CLOCK\\$|NUL|CON|COM\\d|LPT\\d|\\..\*)(\\..+)?$)[^\\x00-\\x1f\\\\?\*:\\"";|/]+$");

#else

QString sPattern("^(([a-zA-Z]:|\\\\)\\\\)?(((\\.)|(\\.\\.)|([^\\\\/:\\\*\\?""\\|<>\\. ](([^\\\\/:\\\*\\?""\\|<>\\. ])|([^\\\\/:\\\*\\?""\\|<>]\*[^\\\\/:\\\*\\?""\\|<>\\. ]))?))\\\\)\*[^\\\\/:\\\*\\?""\\|<>\\. ](([^\\\\/:\\\*\\?""\\|<>\\. ])|([^\\\\/:\\\*\\?""\\|<>]\*[^\\\\/:\\\*\\?""\\|<>\\. ]))?$");

#endif

return QRegExp(sPattern).exactMatch(filename);

}

void MainWindow::doActionSaveFile()

{

if (!haveSelection())

return;

xjHObject clouds("clouds");

xjHObject meshes("meshes");

xjHObject images("images");

xjHObject polylines("polylines");

xjHObject other("other");

xjHObject otherSerializable("serializable");

xjHObject::Container entitiesToDispatch;

entitiesToDispatch.insert(entitiesToDispatch.begin(), m\_selectedEntities.begin(), m\_selectedEntities.end());

xjHObject entitiesToSave;

while (!entitiesToDispatch.empty())

{

xjHObject\* child = entitiesToDispatch.back();

entitiesToDispatch.pop\_back();

if (child->isA(XJ\_TYPES::HIERARCHY\_OBJECT))

{

for (unsigned j = 0; j < child->getChildrenNumber(); ++j)

entitiesToDispatch.push\_back(child->getChild(j));

}

else

{

//we put the entity in the container corresponding to its type

xjHObject\* dest = nullptr;

if (child->isA(XJ\_TYPES::POINT\_CLOUD))

dest = &clouds;

else if (child->isKindOf(XJ\_TYPES::MESH))

dest = &meshes;

else if (child->isKindOf(XJ\_TYPES::IMAGE))

dest = &images;

else if (child->isKindOf(XJ\_TYPES::POLY\_LINE))

dest = &polylines;

else if (child->isSerializable())

dest = &otherSerializable;

else

dest = &other;

assert(dest);

//we don't want double insertions if the user has clicked both the father and child

if (!dest->find(child->getUniqueID()))

{

dest->addChild(child, xjHObject::DP\_NONE);

entitiesToSave.addChild(child, xjHObject::DP\_NONE);

}

}

}

bool hasCloud = (clouds.getChildrenNumber() != 0);

bool hasMesh = (meshes.getChildrenNumber() != 0);

bool hasImages = (images.getChildrenNumber() != 0);

bool hasPolylines = (polylines.getChildrenNumber() != 0);

bool hasSerializable = (otherSerializable.getChildrenNumber() != 0);

bool hasOther = (other.getChildrenNumber() != 0);

int stdSaveTypes = static\_cast<int>(hasCloud)

+ static\_cast<int>(hasMesh)

+ static\_cast<int>(hasImages)

+ static\_cast<int>(hasPolylines)

+ static\_cast<int>(hasSerializable);

if (stdSaveTypes == 0)

{

xjConsole::Error("Can't save selected entity(ies) this way!");

return;

}

//we set up the right file filters, depending on the selected

//entities type (cloud, mesh, etc.).

QStringList fileFilters;

{

for ( const FileIOFilter::Shared &filter : FileIOFilter::GetFilters() )

{

bool atLeastOneExclusive = false;

//can this filter export one or several clouds?

bool canExportClouds = true;

if (hasCloud)

{

bool isExclusive = true;

bool multiple = false;

canExportClouds = ( filter->canSave(XJ\_TYPES::POINT\_CLOUD, multiple, isExclusive)

&& (multiple || clouds.getChildrenNumber() == 1) );

atLeastOneExclusive |= isExclusive;

}

//can this filter export one or several meshes?

bool canExportMeshes = true;

if (hasMesh)

{

bool isExclusive = true;

bool multiple = false;

canExportMeshes = ( filter->canSave(XJ\_TYPES::MESH, multiple, isExclusive)

&& (multiple || meshes.getChildrenNumber() == 1) );

atLeastOneExclusive |= isExclusive;

}

//can this filter export one or several polylines?

bool canExportPolylines = true;

if (hasPolylines)

{

bool isExclusive = true;

bool multiple = false;

canExportPolylines = ( filter->canSave(XJ\_TYPES::POLY\_LINE, multiple, isExclusive)

&& (multiple || polylines.getChildrenNumber() == 1) );

atLeastOneExclusive |= isExclusive;

}

//can this filter export one or several images?

bool canExportImages = true;

if (hasImages)

{

bool isExclusive = true;

bool multiple = false;

canExportImages = ( filter->canSave(XJ\_TYPES::IMAGE, multiple, isExclusive)

&& (multiple || images.getChildrenNumber() == 1) );

atLeastOneExclusive |= isExclusive;

}

//can this filter export one or several other serializable entities?

bool canExportSerializables = true;

if (hasSerializable)

{

//check if all entities have the same type

{

XJ\_CLASS\_ENUM firstClassID = otherSerializable.getChild(0)->getUniqueID();

for (unsigned j = 1; j < otherSerializable.getChildrenNumber(); ++j)

{

if (otherSerializable.getChild(j)->getUniqueID() != firstClassID)

{

//we add a virtual second 'stdSaveType' so as to properly handle exlusivity

++stdSaveTypes;

break;

}

}

}

for (unsigned j = 0; j < otherSerializable.getChildrenNumber(); ++j)

{

xjHObject\* child = otherSerializable.getChild(j);

bool isExclusive = true;

bool multiple = false;

canExportSerializables &= ( filter->canSave(child->getClassID(), multiple, isExclusive)

&& (multiple || otherSerializable.getChildrenNumber() == 1) );

atLeastOneExclusive |= isExclusive;

}

}

bool useThisFilter = canExportClouds

&& canExportMeshes

&& canExportImages

&& canExportPolylines

&& canExportSerializables

&& (!atLeastOneExclusive || stdSaveTypes == 1);

if (useThisFilter)

{

QStringList ff = filter->getFileFilters(false);

for (int j = 0; j < ff.size(); ++j)

fileFilters.append(ff[j]);

}

}

}

//persistent settings

QSettings settings;

settings.beginGroup(xjPS::SaveFile());

//default filter

QString selectedFilter = fileFilters.first();

if (hasCloud)

selectedFilter = settings.value(xjPS::SelectedOutputFilterCloud(),selectedFilter).toString();

else if (hasMesh)

selectedFilter = settings.value(xjPS::SelectedOutputFilterMesh(), selectedFilter).toString();

else if (hasImages)

selectedFilter = settings.value(xjPS::SelectedOutputFilterImage(), selectedFilter).toString();

else if (hasPolylines)

selectedFilter = settings.value(xjPS::SelectedOutputFilterPoly(), selectedFilter).toString();

//default output path (+ filename)

QString currentPath = settings.value(xjPS::CurrentPath(), xjFileUtils::defaultDocPath()).toString();

QString fullPathName = currentPath;

if (haveOneSelection())

{

//hierarchy objects have generally as name: 'filename.ext (fullpath)'

//so we must only take the first part! (otherwise this type of name

//with a path inside perturbs the QFileDialog a lot ;))

QString defaultFileName(m\_selectedEntities.front()->getName());

if (m\_selectedEntities.front()->isA(XJ\_TYPES::HIERARCHY\_OBJECT))

{

QStringList parts = defaultFileName.split(' ', QString::SkipEmptyParts);

if (!parts.empty())

{

defaultFileName = parts[0];

}

}

//we remove the extension

defaultFileName = QFileInfo(defaultFileName).baseName();

if (!IsValidFileName(defaultFileName))

{

xjLog::Warning("[I/O] First entity's name would make an invalid filename! Can't use it...");

defaultFileName = "project";

}

fullPathName += QString("/") + defaultFileName;

}

//ask the user for the output filename

QString selectedFilename = QFileDialog::getSaveFileName(this,

tr("Save file"),

fullPathName,

fileFilters.join(s\_fileFilterSeparator),

&selectedFilter,

XJFileDialogOptions());

if (selectedFilename.isEmpty())

{

//process cancelled by the user

return;

}

//ignored items

if (hasOther)

{

xjConsole::Warning("[I/O] The following selected entities won't be saved:");

for (unsigned i = 0; i < other.getChildrenNumber(); ++i)

{

xjConsole::Warning(QString("\t- %1s").arg(other.getChild(i)->getName()));

}

}

XJ\_FILE\_ERROR result = XJ\_FERR\_NO\_ERROR;

FileIOFilter::SaveParameters parameters;

{

parameters.alwaysDisplaySaveDialog = true;

parameters.parentWidget = this;

}

//specific case: BIN format

if (selectedFilter == BinFilter::GetFileFilter())

{

if ( haveOneSelection() )

{

result = FileIOFilter::SaveToFile(m\_selectedEntities.front(), selectedFilename, parameters, selectedFilter);

}

else

{

//we'll regroup all selected entities in a temporary group

xjHObject tempContainer;

ConvertToGroup(m\_selectedEntities, tempContainer, xjHObject::DP\_NONE);

if (tempContainer.getChildrenNumber())

{

result = FileIOFilter::SaveToFile(&tempContainer, selectedFilename, parameters, selectedFilter);

}

else

{

xjLog::Warning("[I/O] None of the selected entities can be saved this way...");

result = XJ\_FERR\_NO\_SAVE;

}

}

}

else if (entitiesToSave.getChildrenNumber() != 0)

{

//ignored items

//if (hasSerializable)

//{

// if (!hasOther)

// xjConsole::Warning("[I/O] The following selected entites won't be saved:"); //display this warning only if not already done

// for (unsigned i = 0; i < otherSerializable.getChildrenNumber(); ++i)

// xjConsole::Warning(QString("\t- %1").arg(otherSerializable.getChild(i)->getName()));

//}

result = FileIOFilter::SaveToFile( entitiesToSave.getChildrenNumber() > 1 ? &entitiesToSave : entitiesToSave.getChild(0),

selectedFilename,

parameters,

selectedFilter);

if (result == XJ\_FERR\_NO\_ERROR && m\_xjRoot)

{

m\_xjRoot->unselectAllEntities();

}

}

//update default filters

if (hasCloud)

settings.setValue(xjPS::SelectedOutputFilterCloud(),selectedFilter);

if (hasMesh)

settings.setValue(xjPS::SelectedOutputFilterMesh(), selectedFilter);

if (hasImages)

settings.setValue(xjPS::SelectedOutputFilterImage(),selectedFilter);

if (hasPolylines)

settings.setValue(xjPS::SelectedOutputFilterPoly(), selectedFilter);

//we update current file path

currentPath = QFileInfo(selectedFilename).absolutePath();

settings.setValue(xjPS::CurrentPath(),currentPath);

settings.endGroup();

}

void MainWindow::on3DViewActivated(QMdiSubWindow\* mdiWin)

{

xjGLWindow\* win = mdiWin ? GLWindowFromWidget(mdiWin->widget()) : nullptr;

if (win)

{

updateViewModePopUpMenu(win);

updatePivotVisibilityPopUpMenu(win);

m\_UI->actionLockRotationAxis->blockSignals(true);

m\_UI->actionLockRotationAxis->setChecked(win->isRotationAxisLocked());

m\_UI->actionLockRotationAxis->blockSignals(false);

m\_UI->actionEnableStereo->blockSignals(true);

m\_UI->actionEnableStereo->setChecked(win->stereoModeIsEnabled());

m\_UI->actionEnableStereo->blockSignals(false);

m\_UI->actionExclusiveFullScreen->blockSignals(true);

m\_UI->actionExclusiveFullScreen->setChecked(win->exclusiveFullScreen());

m\_UI->actionExclusiveFullScreen->blockSignals(false);

m\_UI->actionShowCursor3DCoordinates->blockSignals(true);

m\_UI->actionShowCursor3DCoordinates->setChecked(win->cursorCoordinatesShown());

m\_UI->actionShowCursor3DCoordinates->blockSignals(false);

m\_UI->actionAutoPickRotationCenter->blockSignals(true);

m\_UI->actionAutoPickRotationCenter->setChecked(win->autoPickPivotAtCenter());

m\_UI->actionAutoPickRotationCenter->blockSignals(false);

}

m\_UI->actionLockRotationAxis->setEnabled(win != nullptr);

m\_UI->actionEnableStereo->setEnabled(win != nullptr);

m\_UI->actionExclusiveFullScreen->setEnabled(win != nullptr);

}

void MainWindow::updateViewModePopUpMenu(xjGLWindow\* win)

{

if (!m\_viewModePopupButton)

return;

//update the view mode pop-up 'top' icon

if (win)

{

bool objectCentered = true;

bool perspectiveEnabled = win->getPerspectiveState(objectCentered);

QAction\* currentModeAction = nullptr;

if (!perspectiveEnabled)

{

currentModeAction = m\_UI->actionSetOrthoView;

}

else if (objectCentered)

{

currentModeAction = m\_UI->actionSetCenteredPerspectiveView;

}

else

{

currentModeAction = m\_UI->actionSetViewerPerspectiveView;

}

assert(currentModeAction);

m\_viewModePopupButton->setIcon(currentModeAction->icon());

m\_viewModePopupButton->setEnabled(true);

}

else

{

m\_viewModePopupButton->setIcon(QIcon());

m\_viewModePopupButton->setEnabled(false);

}

}

void MainWindow::updatePivotVisibilityPopUpMenu(xjGLWindow\* win)

{

if (!m\_pivotVisibilityPopupButton)

return;

//update the pivot visibility pop-up 'top' icon

if (win)

{

QAction\* visibilityAction = nullptr;

switch(win->getPivotVisibility())

{

case xjGLWindow::PIVOT\_HIDE:

visibilityAction = m\_UI->actionSetPivotOff;

break;

case xjGLWindow::PIVOT\_SHOW\_ON\_MOVE:

visibilityAction = m\_UI->actionSetPivotRotationOnly;

break;

case xjGLWindow::PIVOT\_ALWAYS\_SHOW:

visibilityAction = m\_UI->actionSetPivotAlwaysOn;

break;

default:

assert(false);

}

if (visibilityAction)

m\_pivotVisibilityPopupButton->setIcon(visibilityAction->icon());

//pivot is not available in viewer-based perspective!

bool objectCentered = true;

win->getPerspectiveState(objectCentered);

m\_pivotVisibilityPopupButton->setEnabled(objectCentered);

}

else

{

m\_pivotVisibilityPopupButton->setIcon(QIcon());

m\_pivotVisibilityPopupButton->setEnabled(false);

}

}

void MainWindow::updateMenus()

{

xjGLWindow\* active3DView = getActiveGLWindow();

bool hasMdiChild = (active3DView != nullptr);

int mdiChildCount = getGLWindowCount();

bool hasLoadedEntities = (m\_xjRoot && m\_xjRoot->getRootEntity() && m\_xjRoot->getRootEntity()->getChildrenNumber() != 0);

bool hasSelectedEntities = (m\_xjRoot && m\_xjRoot->countSelectedEntities() > 0);

//General Menu

m\_UI->menuEdit->setEnabled(true/\*hasSelectedEntities\*/);

m\_UI->menuTools->setEnabled(true/\*hasSelectedEntities\*/);

//3D Views Menu

m\_UI->actionClose3DView ->setEnabled(hasMdiChild);

m\_UI->actionCloseAll3DViews->setEnabled(mdiChildCount != 0);

m\_UI->actionTile3DViews ->setEnabled(mdiChildCount > 1);

m\_UI->actionCascade3DViews ->setEnabled(mdiChildCount > 1);

m\_UI->actionNext3DView ->setEnabled(mdiChildCount > 1);

m\_UI->actionPrevious3DView ->setEnabled(mdiChildCount > 1);

//Shaders & Filters display Menu

bool shadersEnabled = (active3DView ? active3DView->areShadersEnabled() : false);

m\_UI->actionLoadShader->setEnabled(shadersEnabled);

m\_UI->actionDeleteShader->setEnabled(shadersEnabled);

//View Menu

m\_UI->toolBarView->setEnabled(hasMdiChild);

//oher actions

m\_UI->actionSegment->setEnabled(hasMdiChild && hasSelectedEntities);

m\_UI->actionTranslateRotate->setEnabled(hasMdiChild && hasSelectedEntities);

m\_UI->actionPointPicking->setEnabled(hasMdiChild && hasLoadedEntities);

m\_UI->actionTestFrameRate->setEnabled(hasMdiChild);

m\_UI->actionRenderToFile->setEnabled(hasMdiChild);

m\_UI->actionToggleSunLight->setEnabled(hasMdiChild);

m\_UI->actionToggleCustomLight->setEnabled(hasMdiChild);

m\_UI->actionToggleCenteredPerspective->setEnabled(hasMdiChild);

m\_UI->actionToggleViewerBasedPerspective->setEnabled(hasMdiChild);

//plugins

m\_pluginUIManager->updateMenus();

}

void MainWindow::update3DViewsMenu()

{

m\_UI->menu3DViews->clear();

m\_UI->menu3DViews->addAction(m\_UI->actionNew3DView);

m\_UI->menu3DViews->addSeparator();

m\_UI->menu3DViews->addAction(m\_UI->actionZoomIn);

m\_UI->menu3DViews->addAction(m\_UI->actionZoomOut);

m\_UI->menu3DViews->addSeparator();

m\_UI->menu3DViews->addAction(m\_UI->actionClose3DView);

m\_UI->menu3DViews->addAction(m\_UI->actionCloseAll3DViews);

m\_UI->menu3DViews->addSeparator();

m\_UI->menu3DViews->addAction(m\_UI->actionTile3DViews);

m\_UI->menu3DViews->addAction(m\_UI->actionCascade3DViews);

m\_UI->menu3DViews->addSeparator();

m\_UI->menu3DViews->addAction(m\_UI->actionNext3DView);

m\_UI->menu3DViews->addAction(m\_UI->actionPrevious3DView);

QList<QMdiSubWindow \*> windows = m\_mdiArea->subWindowList();

if (!windows.isEmpty())

{

//Dynamic Separator

QAction\* separator = new QAction(this);

separator->setSeparator(true);

m\_UI->menu3DViews->addAction(separator);

int i = 0;

for ( QMdiSubWindow \*window : windows )

{

xjGLWindow \*child = GLWindowFromWidget(window->widget());

QString text = QString("&%1 %2").arg(++i).arg(child->windowTitle());

QAction \*action = m\_UI->menu3DViews->addAction(text);

action->setCheckable(true);

action->setChecked(child == getActiveGLWindow());

connect(action, &QAction::triggered, this, [=] () {

setActiveSubWindow( window );

} );

}

}

}

void MainWindow::setActiveSubWindow(QWidget \*window)

{

if (!window || !m\_mdiArea)

return;

m\_mdiArea->setActiveSubWindow(qobject\_cast<QMdiSubWindow \*>(window));

}

void MainWindow::redrawAll(bool only2D/\*=false\*/)

{

for ( QMdiSubWindow \*window : m\_mdiArea->subWindowList() )

{

GLWindowFromWidget(window->widget())->redraw(only2D);

}

}

void MainWindow::refreshAll(bool only2D/\*=false\*/)

{

for ( QMdiSubWindow \*window : m\_mdiArea->subWindowList() )

{

GLWindowFromWidget(window->widget())->refresh(only2D);

}

}

void MainWindow::updateUI()

{

updateUIWithSelection();

updateMenus();

updatePropertiesView();

}

MainWindow\* MainWindow::TheInstance()

{

if (!s\_instance)

s\_instance = new MainWindow();

return s\_instance;

}

void MainWindow::DestroyInstance()

{

delete s\_instance;

s\_instance=nullptr;

}

void MainWindow::GetGLWindows(std::vector<xjGLWindow\*>& glWindows)

{

const QList<QMdiSubWindow\*> windows = TheInstance()->m\_mdiArea->subWindowList();

if ( windows.empty() )

return;

glWindows.clear();

glWindows.reserve( windows.size() );

for ( QMdiSubWindow \*window : windows )

{

glWindows.push\_back(GLWindowFromWidget(window->widget()));

}

}

xjGLWindow\* MainWindow::GetActiveGLWindow()

{

return TheInstance()->getActiveGLWindow();

}

xjGLWindow\* MainWindow::GetGLWindow(const QString& title)

{

const QList<QMdiSubWindow \*> windows = TheInstance()->m\_mdiArea->subWindowList();

if ( windows.empty() )

return nullptr;

for ( QMdiSubWindow \*window : windows )

{

xjGLWindow\* win = GLWindowFromWidget(window->widget());

if (win->windowTitle() == title)

return win;

}

return nullptr;

}

void MainWindow::RefreshAllGLWindow(bool only2D/\*=false\*/)

{

TheInstance()->refreshAll(only2D);

}

void MainWindow::UpdateUI()

{

TheInstance()->updateUI();

}

xjDBRoot\* MainWindow::db()

{

return m\_xjRoot;

}

void MainWindow::addEditPlaneAction( QMenu &menu ) const

{

menu.addAction( m\_UI->actionEditPlane );

}

xjHObject\* MainWindow::dbRootObject()

{

return (m\_xjRoot ? m\_xjRoot->getRootEntity() : nullptr);

}

xjUniqueIDGenerator::Shared MainWindow::getUniqueIDGenerator()

{

return xjObject::GetUniqueIDGenerator();

}

void MainWindow::createGLWindow(xjGLWindow\*& window, QWidget\*& widget) const

{

bool stereoMode = QSurfaceFormat::defaultFormat().stereo();

CreateGLWindow(window, widget, stereoMode);

assert(window && widget);

}

void MainWindow::destroyGLWindow(xjGLWindow\* view3D) const

{

if (view3D)

{

view3D->setParent(nullptr);

delete view3D;

}

}

/\* PCA: Principal Component Analysis 主成分分析 \*/

void MainWindow::on\_actionPCA\_triggered()

{

xjProgressDialog pDlg(false, this);

pDlg.setAutoClose(true);

pDlg.setMethodTitle("...");

pDlg.start();

xjAlgorithm \*xjal = new xjAlgorithm();

QStringList xjList; xjList.clear();

xjHObject::Container selectedEntities = getSelectedEntities(); //warning, getSelectedEntites may change during this loop!

xjHObject\* resultContainer = new xjHObject("PCA");

for (xjHObject \*entity : selectedEntities)

{

if (entity->isKindOf(XJ\_TYPES::POINT\_CLOUD))

{

xjPointCloud\* cloud = xjHObjectCaster::ToPointCloud(entity);

if (cloud)

{

/\* To do... \*/

QString cloudName = cloud->getName();

if (cloudName.contains(" - Cloud"))

cloudName = cloudName.left(cloudName.length() - 8);

/\* compute PCA \*/

Eigen::Matrix4d matPCA = xjal->xjComputePCA(cloud);

/\* apply PCA \*/

xjPointCloud \*pcR = xjal->xjApplyPCA(cloud, matPCA);

/\* xj: eigenValues \*/

xjLog::Print("eigenValues: " + QString::number(matPCA(3,0)) + ", " + QString::number(matPCA(3, 1)) + ", " + QString::number(matPCA(3, 2)));

xjLog::Print("eigenVector:");

xjLog::Print(QString::number(0) + ": " + QString::number(matPCA(0, 0)) + ", " + QString::number(matPCA(1,0)) + ", " + QString::number(matPCA(2, 0)));

xjLog::Print(QString::number(1) + ": " + QString::number(matPCA(0, 1)) + ", " + QString::number(matPCA(1,1)) + ", " + QString::number(matPCA(2, 1)));

xjLog::Print(QString::number(2) + ": " + QString::number(matPCA(0, 2)) + ", " + QString::number(matPCA(1,2)) + ", " + QString::number(matPCA(2, 2)));

}

}

}

addToDB(xjList,0, 0);

refreshAll();

updateUI();

delete xjal;

xjal = nullptr;

xjLog::Print("[PCA] OK");

}

void MainWindow::closeAll()

{

if (!m\_xjRoot)

{

return;

}

QMessageBox message\_box(QMessageBox::Question,

tr("Close all"),

tr("Are you sure you want to remove all loaded entities?"),

QMessageBox::Yes | QMessageBox::No,

this);

if (message\_box.exec() == QMessageBox::No)

{

return;

}

m\_xjRoot->unloadAll();

redrawAll(false);

}