#include "mainwindow.h"

//CCLib Includes

#include <CloudSamplingTools.h>

#include <Delaunay2dMesh.h>

#include <Jacobi.h>

#include <MeshSamplingTools.h>

#include <NormalDistribution.h>

#include <ParallelSort.h>

#include <PointCloud.h>

#include <ScalarFieldTools.h>

#include <StatisticalTestingTools.h>

#include <WeibullDistribution.h>

//for tests

#include <ChamferDistanceTransform.h>

#include <SaitoSquaredDistanceTransform.h>

//qCC\_db

#include <d2DLabel.h>

#include <d2DViewportObject.h>

#include <dCameraSensor.h>

#include <dColorScalesManager.h>

#include <dFacet.h>

#include <dFileUtils.h>

#include <dGBLSensor.h>

#include <dImage.h>

#include <dKdTree.h>

#include <dPlane.h>

#include <dProgressDialog.h>

#include <dQuadric.h>

#include <dSphere.h>

#include <dCylinder.h>

#include <dSubMesh.h>

//qCC\_io

#include <dShiftAndScaleCloudDlg.h>

#include <BinFilter.h>

#include <DepthMapFileFilter.h>

//QCC\_glWindow

#include <dGLWidget.h>

#include <dRenderingTools.h>

//local includes

#include "dConsole.h"

#include "dEntityAction.h"

#include "dHistogramWindow.h"

#include "dInnerRect2DFinder.h"

//common

#include <dPickingHub.h>

//common dialogs

#include <dCameraParamEditDlg.h>

#include <dDisplayOptionsDlg.h>

#include <dPickOneElementDlg.h>

#include <dStereoModeDlg.h>

//dialogs

#include "dAboutDialog.h"

#include "dAdjustZoomDlg.h"

#include "dAlignDlg.h" //Aurelien BEY

#include "dApplication.h"

#include "dApplyTransformationDlg.h"

#include "dAskThreeDoubleValuesDlg.h"

#include "dBoundingBoxEditorDlg.h"

#include "dCamSensorProjectionDlg.h"

#include "dClippingBoxTool.h"

#include "dColorFromScalarDlg.h"

#include "dColorScaleEditorDlg.h"

#include "dComparisonDlg.h"

#include "dPrimitiveDistanceDlg.h"

#include "dFilterByValueDlg.h"

#include "dGBLSensorProjectionDlg.h"

#include "dGeomFeaturesDlg.h"

#include "dGraphicalSegmentationTool.h"

#include "dGraphicalTransformationTool.h"

#include "dItemSelectionDlg.h"

#include "dLabelingDlg.h"

#include "dMatchScalesDlg.h"

#include "dNoiseFilterDlg.h"

#include "dOrderChoiceDlg.h"

#include "dPlaneEditDlg.h"

#include "dPointListPickingDlg.h"

#include "dPointPairRegistrationDlg.h"

#include "dPointPropertiesDlg.h" //Aurelien BEY

#include "dPrimitiveFactoryDlg.h"

#include "dPtsSamplingDlg.h"

#include "dRasterizeTool.h"

#include "dRegistrationDlg.h" //Aurelien BEY

#include "dRenderToFileDlg.h"

#include "dScaleDlg.h"

#include "dSectionExtractionTool.h"

#include "dSensorComputeDistancesDlg.h"

#include "dSensorComputeScatteringAnglesDlg.h"

#include "dSORFilterDlg.h"

#include "dSubsamplingDlg.h" //Aurelien BEY

#include "dTracePolylineTool.h"

#include "dTranslationManager.h"

#include "dUnrollDlg.h"

#include "dVolumeCalcTool.h"

#include "dWaveformDialog.h"

#include "dEntitySelectionDlg.h"

//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 dGLWindow\* s\_pickingWindow = nullptr;

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

static std::vector<d2DLabel\*> s\_levelLabels;

static dPointCloud\* s\_levelMarkersCloud = nullptr;

static dHObject\* s\_levelEntity = nullptr;

static QFileDialog::Options CCFileDialogOptions()

{

//dialog options

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

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

{

dialogOptions |= QFileDialog::DontUseNativeDialog;

}

return dialogOptions;

}

MainWindow::MainWindow()

: m\_UI( new Ui::MainWindow )

, m\_dRoot(nullptr)

, m\_uiFrozen(false)

, m\_recentFiles(new dRecentFiles(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("点云去噪软件 v1.0"));

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

dTranslationManager::get().populateMenu( m\_UI->menuLanguage, dApp->translationPath() );

#ifdef Q\_OS\_MAC

m\_UI->actionAbout->setMenuRole( QAction::AboutRole );

m\_UI->actionAboutPlugins->setMenuRole( QAction::ApplicationSpecificRole );

m\_UI->actionFullScreen->setText( tr( "Enter Full Screen" ) );

m\_UI->actionFullScreen->setShortcut( QKeySequence( Qt::CTRL + Qt::META + Qt::Key\_F ) );

#endif

// Set up dynamic menus

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

//Console

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

m\_UI->actionEnableQtWarnings->setChecked(dConsole::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\_dRoot = new dDBRoot(m\_UI->dbTreeView, m\_UI->propertiesTreeView, this);

connect(m\_dRoot, &dDBRoot::selectionChanged, this, &MainWindow::updateUIWithSelection);

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

connect(m\_dRoot, &dDBRoot::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 dPickingHub(this, this);

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

}

connectActions();

new3DView(true);

setupInputDevices();

freezeUI(false);

updateUI();

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

#ifdef USE\_TBB

dConsole::Print( QStringLiteral( "[TBB] Using Intel's Threading Building Blocks %1.%2" )

.arg( QString::number( TBB\_VERSION\_MAJOR ), QString::number( TBB\_VERSION\_MINOR ) ) );

#endif

dConsole::Print("CloudCompare started!");

}

MainWindow::~MainWindow()

{

destroyInputDevices();

cancelPreviousPickingOperation(false); //just in case

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

m\_dRoot->disconnect();

m\_mdiArea->disconnect();

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

dDBRoot\* dRoot = m\_dRoot;

m\_dRoot = 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\_dRoot)

dRoot->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())

{

dMDIDialogs 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 (dRoot)

{

delete dRoot;

dRoot = nullptr;

}

delete m\_UI;

m\_UI = nullptr;

dConsole::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() );

//for ( QToolBar \*toolbar : m\_pluginUIManager->additionalPluginToolbars() )

//{

// addToolBar( Qt::TopToolBarArea, toolbar );

//}

//// Set up dynamic menus

//m\_UI->menubar->insertMenu( m\_UI->menu3DViews->menuAction(), m\_pluginUIManager->pluginMenu() );

//m\_UI->menuDisplay->insertMenu( m\_UI->menuActiveScalarField->menuAction(), m\_pluginUIManager->shaderAndFilterMenu() );

//m\_UI->menuToolbars->addAction( m\_pluginUIManager->actionShowMainPluginToolbar() );

//m\_UI->menuToolbars->addAction( m\_pluginUIManager->actionShowGLFilterToolbar() );

}

void MainWindow::doEnableQtWarnings(bool state)

{

dConsole::EnableQtMessages(state);

}

static double s\_kdTreeMaxErrorPerCell = 0.1;

void MainWindow::doActionComputeKdTree()

{

dGenericPointCloud\* cloud = nullptr;

if ( haveOneSelection() )

{

dHObject\* ent = m\_selectedEntities.back();

bool lockedVertices;

cloud = dHObjectCaster::ToGenericPointCloud(ent,&lockedVertices);

if (lockedVertices)

{

dUtils::DisplayLockedVerticesWarning(ent->getName(),true);

return;

}

}

if (!cloud)

{

dLog::Error("Selected one and only one point cloud or mesh!");

return;

}

bool ok;

s\_kdTreeMaxErrorPerCell = QInputDialog::getDouble(this, "Compute Kd-tree", "Max error per leaf cell:", s\_kdTreeMaxErrorPerCell, 1.0e-6, 1.0e6, 6, &ok);

if (!ok)

return;

dProgressDialog pDlg(true, this);

//computation

QElapsedTimer eTimer;

eTimer.start();

dKdTree\* kdtree = new dKdTree(cloud);

if (kdtree->build(s\_kdTreeMaxErrorPerCell, CCLib::DistanceComputationTools::MAX\_DIST\_95\_PERCENT, 4, 1000, &pDlg))

{

qint64 elapsedTime\_ms = eTimer.elapsed();

dConsole::Print("[doActionComputeKdTree] Timing: %2.3f s",static\_cast<double>(elapsedTime\_ms)/1.0e3);

cloud->setEnabled(true); //for mesh vertices!

cloud->addChild(kdtree);

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

kdtree->setVisible(true);

kdtree->prepareDisplayForRefresh();

#ifdef QT\_DEBUG

kdtree->convertCellIndexToSF();

#else

kdtree->convertCellIndexToRandomColor();

#endif

addToDB(kdtree);

refreshAll();

updateUI();

}

else

{

dLog::Error("An error odurred!");

delete kdtree;

kdtree = nullptr;

}

}

void MainWindow::doActionComputeOctree()

{

if ( !dEntityAction::computeOctree(m\_selectedEntities, this) )

return;

refreshAll();

updateUI();

}

void MainWindow::zoomOn(dHObject\* object)

{

dGLWindow\* win = static\_cast<dGLWindow\*>(object->getDisplay());

if (win)

{

dBBox box = object->getDisplayBB\_recursive(false,win);

win->updateConstellationCenterAndZoom(&box);

}

}

void MainWindow::doActionRegister()

{

if ( m\_selectedEntities.size() != 2

|| (!m\_selectedEntities[0]->isKindOf(CC\_TYPES::POINT\_CLOUD) && !m\_selectedEntities[0]->isKindOf(CC\_TYPES::MESH))

|| (!m\_selectedEntities[1]->isKindOf(CC\_TYPES::POINT\_CLOUD) && !m\_selectedEntities[1]->isKindOf(CC\_TYPES::MESH)) )

{

dConsole::Error("Select 2 point clouds or meshes!");

return;

}

dHObject\* data = static\_cast<dHObject\*>(m\_selectedEntities[1]);

dHObject\* model = static\_cast<dHObject\*>(m\_selectedEntities[0]);

dRegistrationDlg rDlg(data,model,this);

if (!rDlg.exec())

return;

//DGM (23/01/09): model and data order may have changed!

model = rDlg.getModelEntity();

data = rDlg.getDataEntity();

double minRMSDecrease = rDlg.getMinRMSDecrease();

unsigned maxIterationCount = rDlg.getMaxIterationCount();

unsigned randomSamplingLimit = rDlg.randomSamplingLimit();

bool removeFarthestPoints = rDlg.removeFarthestPoints();

bool useDataSFAsWeights = rDlg.useDataSFAsWeights();

bool useModelSFAsWeights = rDlg.useModelSFAsWeights();

bool adjustScale = rDlg.adjustScale();

int transformationFilters = rDlg.getTransformationFilters();

unsigned finalOverlap = rDlg.getFinalOverlap();

CCLib::ICPRegistrationTools::CONVERGENCE\_TYPE method = rDlg.getConvergenceMethod();

int maxThreadCount = rDlg.getMaxThreadCount();

//semi-persistent storage (for next call)

rDlg.saveParameters();

dGLMatrix transMat;

double finalError = 0.0;

double finalScale = 1.0;

unsigned finalPointCount = 0;

if (dRegistrationTools::ICP( data,

model,

transMat,

finalScale,

finalError,

finalPointCount,

minRMSDecrease,

maxIterationCount,

randomSamplingLimit,

removeFarthestPoints,

method,

adjustScale,

finalOverlap / 100.0,

useDataSFAsWeights,

useModelSFAsWeights,

transformationFilters,

maxThreadCount,

this))

{

QString rmsString = QString("Final RMS: %1 (computed on %2 points)").arg(finalError).arg(finalPointCount);

dLog::Print(QString("[Register] ") + rmsString);

QStringList summary;

summary << rmsString;

summary << "----------------";

//transformation matrix

{

summary << "Transformation matrix";

summary << transMat.toString(3, '\t'); //low precision, just for display

summary << "----------------";

dLog::Print("[Register] Applied transformation matrix:");

dLog::Print(transMat.toString(12, ' ')); //full precision

dLog::Print("Hint: copy it (CTRL+C) and apply it - or its inverse - on any entity with the 'Edit > Apply transformation' tool");

}

if (adjustScale)

{

QString scaleString = QString("Scale: %1 (already integrated in above matrix!)").arg(finalScale);

dLog::Warning(QString("[Register] ") + scaleString);

summary << scaleString;

}

else

{

dLog::Print(QString("[Register] Scale: fixed (1.0)"));

summary << "Scale: fixed (1.0)";

}

//overlap

summary << "----------------";

QString overlapString = QString("Theoretical overlap: %1%").arg(finalOverlap);

dLog::Print(QString("[Register] ") + overlapString);

summary << overlapString;

summary << "----------------";

summary << "This report has been output to Console (F8)";

//cloud to move

dGenericPointCloud\* pc = nullptr;

if (data->isKindOf(CC\_TYPES::POINT\_CLOUD))

{

pc = dHObjectCaster::ToGenericPointCloud(data);

}

else if (data->isKindOf(CC\_TYPES::MESH))

{

dGenericMesh\* mesh = dHObjectCaster::ToGenericMesh(data);

pc = mesh->getAssociatedCloud();

//warning: point cloud is locked!

if (pc->isLocked())

{

pc = nullptr;

//we ask the user about cloning the 'data' mesh

QMessageBox::StandardButton result = QMessageBox::question( this,

"Registration",

"Data mesh vertices are locked (they may be shared with other meshes): Do you wish to clone this mesh to apply transformation?",

QMessageBox::Ok | QMessageBox::Cancel,

QMessageBox::Ok);

//continue process?

if (result == QMessageBox::Ok)

{

dGenericMesh\* newMesh = nullptr;

if (mesh->isA(CC\_TYPES::MESH))

newMesh = static\_cast<dMesh\*>(mesh)->cloneMesh();

else

{

//FIXME TODO

dLog::Error("Doesn't work on sub-meshes yet!");

}

if (newMesh)

{

newMesh->setDisplay(data->getDisplay());

addToDB(newMesh);

data = newMesh;

pc = newMesh->getAssociatedCloud();

}

else

{

dLog::Error("Failed to clone 'data' mesh! (not enough memory?)");

}

}

}

}

//if we managed to get a point cloud to move!

if (pc)

{

//we temporarily detach cloud, as it may undergo

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

dHObjectContext objContext = removeObjectTemporarilyFromDBTree(pc);

pc->applyRigidTransformation(transMat);

putObjectBackIntoDBTree(pc,objContext);

//don't forget to update mesh bounding box also!

if (data->isKindOf(CC\_TYPES::MESH))

dHObjectCaster::ToGenericMesh(data)->refreshBB();

//don't forget global shift

dGenericPointCloud\* refPc = dHObjectCaster::ToGenericPointCloud(model);

if (refPc)

{

if (refPc->isShifted())

{

const CCVector3d& Pshift = refPc->getGlobalShift();

const double& scale = refPc->getGlobalScale();

pc->setGlobalShift(Pshift);

pc->setGlobalScale(scale);

dLog::Warning(QString("[ICP] Aligned entity global shift has been updated to match the reference: (%1,%2,%3) [x%4]").arg(Pshift.x).arg(Pshift.y).arg(Pshift.z).arg(scale));

}

else if (pc->isShifted()) //we'll ask the user first before dropping the shift information on the aligned cloud

{

if (QMessageBox::question(this, "Drop shift information?", "Aligned entity is shifted but reference cloud is not: drop global shift information?", QMessageBox::Yes, QMessageBox::No) == QMessageBox::Yes)

{

pc->setGlobalShift(0,0,0);

pc->setGlobalScale(1.0);

dLog::Warning(QString("[ICP] Aligned entity global shift has been reset to match the reference!"));

}

}

}

data->prepareDisplayForRefresh\_recursive();

data->setName(data->getName()+QString(".registered"));

zoomOn(data);

}

//pop-up summary

QMessageBox::information(this, "Register info", summary.join("\n"));

forceConsoleDisplay();

}

refreshAll();

updateUI();

}

void MainWindow::doActionSubsample()

{

//find candidates

std::vector<dPointCloud\*> clouds;

unsigned maxPointCount = 0;

double maxCloudRadius = 0;

ScalarType sfMin = NAN\_VALUE;

ScalarType sfMax = NAN\_VALUE;

{

for ( dHObject \*entity : getSelectedEntities() )

{

if (entity->isA(CC\_TYPES::POINT\_CLOUD))

{

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

clouds.push\_back(cloud);

maxPointCount = std::max<unsigned>(maxPointCount, cloud->size());

maxCloudRadius = std::max<double>(maxCloudRadius, cloud->getOwnBB().getDiagNorm());

//we also look for the min and max sf values

dScalarField\* sf = cloud->getCurrentDisplayedScalarField();

if (sf)

{

if (!dScalarField::ValidValue(sfMin) || sfMin > sf->getMin())

sfMin = sf->getMin();

if (!dScalarField::ValidValue(sfMax) || sfMax < sf->getMax())

sfMax = sf->getMax();

}

}

}

}

if (clouds.empty())

{

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

return;

}

//Display dialog

dSubsamplingDlg sDlg(maxPointCount, maxCloudRadius, this);

bool hasValidSF = dScalarField::ValidValue(sfMin) && dScalarField::ValidValue(sfMax);

if (hasValidSF)

sDlg.enableSFModulation(sfMin,sfMax);

if (!sDlg.exec())

return;

//process clouds

dHObject::Container resultingClouds;

{

dProgressDialog pDlg(false, this);

pDlg.setAutoClose(false);

pDlg.setMethodTitle(tr("Subsampling"));

bool errors = false;

QElapsedTimer eTimer;

eTimer.start();

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

{

dPointCloud\* cloud = clouds[i];

CCLib::ReferenceCloud \*sampledCloud = sDlg.getSampledCloud(cloud,&pDlg);

if (!sampledCloud)

{

dConsole::Warning(QString("[Subsampling] Failed to subsample cloud '%1'!").arg(cloud->getName()));

errors = true;

continue;

}

int warnings = 0;

dPointCloud \*newPointCloud = cloud->partialClone(sampledCloud,&warnings);

delete sampledCloud;

sampledCloud = nullptr;

if (newPointCloud)

{

newPointCloud->setName(cloud->getName() + QString(".subsampled"));

newPointCloud->setGlobalShift(cloud->getGlobalShift());

newPointCloud->setGlobalScale(cloud->getGlobalScale());

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

newPointCloud->prepareDisplayForRefresh();

if (cloud->getParent())

cloud->getParent()->addChild(newPointCloud);

cloud->setEnabled(false);

addToDB(newPointCloud);

newPointCloud->prepareDisplayForRefresh();

resultingClouds.push\_back(newPointCloud);

if (warnings)

{

dLog::Warning("[Subsampling] Not enough memory: colors, normals or scalar fields may be missing!");

errors = true;

}

}

else

{

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

break;

}

}

dLog::Print("[Subsampling] Timing: %3.3f s.",eTimer.elapsed()/1000.0);

if (errors)

{

dLog::Error("Errors odurred (see console)");

}

}

if (m\_dRoot)

m\_dRoot->selectEntities(resultingClouds);

refreshAll();

updateUI();

}

void MainWindow::doActionLabelConnectedComponents()

{

//keep only the point clouds!

std::vector<dGenericPointCloud\*> clouds;

{

for ( dHObject \*entity : getSelectedEntities() )

{

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

clouds.push\_back(dHObjectCaster::ToGenericPointCloud(entity));

}

}

size\_t count = clouds.size();

if (count == 0)

return;

dLabelingDlg dlg(this);

if (count == 1)

dlg.octreeLevelSpinBox->setCloud(clouds.front());

if (!dlg.exec())

return;

int octreeLevel = dlg.getOctreeLevel();

unsigned minComponentSize = static\_cast<unsigned>(std::max(0, dlg.getMinPointsNb()));

bool randColors = dlg.randomColors();

dProgressDialog pDlg(false, this);

pDlg.setAutoClose(false);

//we unselect all entities as we are going to automatically select the created components

//(otherwise the user won't perceive the change!)

if (m\_dRoot)

{

m\_dRoot->unselectAllEntities();

}

for ( dGenericPointCloud \*cloud : clouds )

{

if (cloud && cloud->isA(CC\_TYPES::POINT\_CLOUD))

{

dPointCloud\* pc = static\_cast<dPointCloud\*>(cloud);

dOctree::Shared theOctree = cloud->getOctree();

if (!theOctree)

{

dProgressDialog pOctreeDlg(true, this);

theOctree = cloud->computeOctree(&pOctreeDlg);

if (!theOctree)

{

dConsole::Error(QString("Couldn't compute octree for cloud '%1'!").arg(cloud->getName()));

break;

}

}

//we create/activate CCs label scalar field

int sfIdx = pc->getScalarFieldIndexByName(CC\_CONNECTED\_COMPONENTS\_DEFAULT\_LABEL\_NAME);

if (sfIdx < 0)

{

sfIdx = pc->addScalarField(CC\_CONNECTED\_COMPONENTS\_DEFAULT\_LABEL\_NAME);

}

if (sfIdx < 0)

{

dConsole::Error("Couldn't allocate a new scalar field for computing CC labels! Try to free some memory ...");

break;

}

pc->setCurrentScalarField(sfIdx);

//we try to label all CCs

CCLib::ReferenceCloudContainer components;

int componentCount = CCLib::AutoSegmentationTools::labelConnectedComponents(cloud,

static\_cast<unsigned char>(octreeLevel),

false,

&pDlg,

theOctree.data());

if (componentCount >= 0)

{

//if sudessful, we extract each CC (stored in "components")

//safety test

int realComponentCount = 0;

{

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

{

if (components[i]->size() >= minComponentSize)

{

++realComponentCount;

}

}

}

if (realComponentCount > 500)

{

//too many components

if (QMessageBox::warning(this, "Many components", QString("Do you really expect up to %1 components?\n(this may take a lot of time to process and display)").arg(realComponentCount), QMessageBox::Yes, QMessageBox::No) == QMessageBox::No)

{

//cancel

pc->deleteScalarField(sfIdx);

if (pc->getNumberOfScalarFields() != 0)

{

pc->setCurrentDisplayedScalarField(static\_cast<int>(pc->getNumberOfScalarFields()) - 1);

}

else

{

pc->showSF(false);

}

pc->prepareDisplayForRefresh();

continue;

}

}

pc->getCurrentInScalarField()->computeMinAndMax();

if (!CCLib::AutoSegmentationTools::extractConnectedComponents(cloud, components))

{

dConsole::Warning(QString("[doActionLabelConnectedComponents] Something went wrong while extracting CCs from cloud %1...").arg(cloud->getName()));

}

}

else

{

dConsole::Warning(QString("[doActionLabelConnectedComponents] Something went wrong while extracting CCs from cloud %1...").arg(cloud->getName()));

}

//we delete the CCs label scalar field (we don't need it anymore)

pc->deleteScalarField(sfIdx);

sfIdx = -1;

//we create "real" point clouds for all CCs

if (!components.empty())

{

createComponentsClouds(cloud, components, minComponentSize, randColors, true);

}

}

}

refreshAll();

updateUI();

}

void MainWindow::doActionDeleteScanGrids()

{

//look for clouds with scan grids

for ( dHObject \*entity : getSelectedEntities() )

{

if (!entity || !entity->isA(CC\_TYPES::POINT\_CLOUD))

{

continue;

}

dPointCloud\* cloud = dHObjectCaster::ToPointCloud(entity);

assert(cloud);

if(cloud->gridCount() > 0)

{

cloud->removeGrids();

}

}

refreshAll();

updateUI();

}

void MainWindow::doActionMeshScanGrids()

{

//ask the user for the min angle (inside triangles)

static double s\_meshMinTriangleAngle\_deg = 1.0;

{

bool ok = true;

double minAngle\_deg = QInputDialog::getDouble(this, "Triangulate", "Min triangle angle (in degrees)", s\_meshMinTriangleAngle\_deg, 0, 90.0, 3, &ok);

if (!ok)

return;

s\_meshMinTriangleAngle\_deg = minAngle\_deg;

}

//look for clouds with scan grids

for ( dHObject \*entity : getSelectedEntities() )

{

if (!entity || !entity->isA(CC\_TYPES::POINT\_CLOUD))

{

continue;

}

dPointCloud\* cloud = dHObjectCaster::ToPointCloud(entity);

assert(cloud);

for (size\_t i = 0; i < cloud->gridCount(); ++i)

{

dPointCloud::Grid::Shared grid = cloud->grid(i);

if (!grid)

{

assert(false);

continue;

}

dMesh\* gridMesh = cloud->triangulateGrid(\*grid, s\_meshMinTriangleAngle\_deg);

if (gridMesh)

{

cloud->addChild(gridMesh);

cloud->setVisible(false); //hide the cloud

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

addToDB(gridMesh, false, true, false, false);

gridMesh->prepareDisplayForRefresh();

}

}

}

refreshAll();

updateUI();

}

void MainWindow::doActionComputeMeshAA()

{

doActionComputeMesh(DELAUNAY\_2D\_AXIS\_ALIGNED);

}

void MainWindow::doActionComputeMeshLS()

{

doActionComputeMesh(DELAUNAY\_2D\_BEST\_LS\_PLANE);

}

void MainWindow::doActionComputeMesh(CC\_TRIANGULATION\_TYPES type)

{

//ask the user for the max edge length

static double s\_meshMaxEdgeLength = 0.0;

{

bool ok = true;

double maxEdgeLength = QInputDialog::getDouble(this, "Triangulate", "Max edge length (0 = no limit)", s\_meshMaxEdgeLength, 0, 1.0e9, 8, &ok);

if (!ok)

return;

s\_meshMaxEdgeLength = maxEdgeLength;

}

//select candidates

dHObject::Container clouds;

bool hadNormals = false;

{

for ( dHObject \*entity : getSelectedEntities() )

{

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

{

clouds.push\_back(entity);

if (entity->isA(CC\_TYPES::POINT\_CLOUD))

{

hadNormals |= static\_cast<dPointCloud\*>(entity)->hasNormals();

}

}

}

}

//if the cloud(s) already had normals, ask the use if wants to update them or keep them as is (can look strange!)

bool updateNormals = false;

if (hadNormals)

{

updateNormals = (QMessageBox::question( this,

"Keep old normals?",

"Cloud(s) already have normals. Do you want to update them (yes) or keep the old ones (no)?",

QMessageBox::Yes,

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

}

dProgressDialog pDlg(false, this);

pDlg.setAutoClose(false);

pDlg.setWindowTitle(tr("Triangulation"));

pDlg.setInfo(tr("Triangulation in progress..."));

pDlg.setRange(0, 0);

pDlg.show();

QApplication::processEvents();

bool errors = false;

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

{

dHObject\* ent = clouds[i];

assert(ent->isKindOf(CC\_TYPES::POINT\_CLOUD));

//compute mesh

dGenericPointCloud\* cloud = dHObjectCaster::ToGenericPointCloud(ent);

dMesh\* mesh = dMesh::Triangulate( cloud,

type,

updateNormals,

static\_cast<PointCoordinateType>(s\_meshMaxEdgeLength),

2 //XY plane by default

);

if (mesh)

{

cloud->setVisible(false); //can't disable the cloud as the resulting mesh will be its child!

cloud->addChild(mesh);

cloud->prepareDisplayForRefresh\_recursive();

addToDB(mesh);

if (i == 0)

{

m\_dRoot->selectEntity(mesh); //auto-select first element

}

}

else

{

errors = true;

}

}

if (errors)

{

dConsole::Error("Error(s) odurred! See the Console messages");

}

refreshAll();

updateUI();

}

void MainWindow::doActionFitQuadric()

{

bool errors = false;

//for all selected entities

for ( dHObject \*entity : getSelectedEntities() )

{

//look for clouds

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

{

dGenericPointCloud\* cloud = dHObjectCaster::ToGenericPointCloud(entity);

double rms = 0.0;

dQuadric\* quadric = dQuadric::Fit(cloud,&rms);

if (quadric)

{

cloud->addChild(quadric);

quadric->setName(QString("Quadric (%1)").arg(cloud->getName()));

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

quadric->prepareDisplayForRefresh();

addToDB(quadric);

dConsole::Print(QString("[doActionFitQuadric] Quadric local coordinate system:"));

dConsole::Print(quadric->getTransformation().toString(12,' ')); //full precision

dConsole::Print(QString("[doActionFitQuadric] Quadric equation (in local coordinate system): ") + quadric->getEquationString());

dConsole::Print(QString("[doActionFitQuadric] RMS: %1").arg(rms));

#if 0

//test: project the input cloud on the quadric

if (cloud->isA(CC\_TYPES::POINT\_CLOUD))

{

dPointCloud\* newCloud = static\_cast<dPointCloud\*>(cloud)->cloneThis();

if (newCloud)

{

const PointCoordinateType\* eq = quadric->getEquationCoefs();

const Tuple3ub& dims = quadric->getEquationDims();

const unsigned char dX = dims.x;

const unsigned char dY = dims.y;

const unsigned char dZ = dims.z;

const dGLMatrix& trans = quadric->getTransformation();

dGLMatrix invTrans = trans.inverse();

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

{

CCVector3\* P = const\_cast<CCVector3\*>(newCloud->getPoint(i));

CCVector3 Q = invTrans \* (\*P);

Q.u[dZ] = eq[0] + eq[1]\*Q.u[dX] + eq[2]\*Q.u[dY] + eq[3]\*Q.u[dX]\*Q.u[dX] + eq[4]\*Q.u[dX]\*Q.u[dY] + eq[5]\*Q.u[dY]\*Q.u[dY];

\*P = trans \* Q;

}

newCloud->invalidateBoundingBox();

newCloud->setName(newCloud->getName() + ".projection\_on\_quadric");

addToDB(newCloud);

}

}

#endif

}

else

{

dConsole::Warning(QString("Failed to compute quadric on cloud '%1'").arg(cloud->getName()));

errors = true;

}

}

}

if (errors)

{

dConsole::Error("Error(s) odurred: see console");

}

refreshAll();

}

void MainWindow::doActionComputeDistanceMap()

{

static unsigned steps = 128;

static double margin = 0.0;

static bool filterRange = false;

static double range[2] = { 0.0, 1.0 };

//show dialog

{

QDialog dialog(this);

Ui\_DistanceMapDialog ui;

ui.setupUi(&dialog);

ui.stepsSpinBox->setValue(static\_cast<int>(steps));

ui.marginDoubleSpinBox->setValue(margin);

ui.rangeCheckBox->setChecked(filterRange);

ui.minDistDoubleSpinBox->setValue(range[0]);

ui.maxDistDoubleSpinBox->setValue(range[1]);

if (!dialog.exec())

{

return;

}

steps = static\_cast<unsigned>(ui.stepsSpinBox->value());

margin = ui.marginDoubleSpinBox->value();

filterRange = ui.rangeCheckBox->isChecked();

range[0] = ui.minDistDoubleSpinBox->value();

range[1] = ui.maxDistDoubleSpinBox->value();

}

dProgressDialog pDlg(true, this);

pDlg.setAutoClose(false);

for ( dHObject \*entity : getSelectedEntities() )

{

if (!entity->isKindOf(CC\_TYPES::MESH) && !entity->isKindOf(CC\_TYPES::POINT\_CLOUD))

{

//non handled entity type

continue;

}

//CCLib::ChamferDistanceTransform cdt;

CCLib::SaitoSquaredDistanceTransform cdt;

if (!cdt.initGrid(Tuple3ui(steps, steps, steps)))

{

//not enough memory

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

return;

}

dBBox box = entity->getOwnBB();

PointCoordinateType largestDim = box.getMaxBoxDim() + static\_cast<PointCoordinateType>(margin);

PointCoordinateType cellDim = largestDim / steps;

CCVector3 minCorner = box.getCenter() - CCVector3(1, 1, 1) \* (largestDim / 2);

bool result = false;

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

{

dMesh\* mesh = static\_cast<dMesh\*>(entity);

result = cdt.initDT(mesh, cellDim, minCorner, &pDlg);

}

else

{

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

result = cdt.initDT(cloud, cellDim, minCorner, &pDlg);

}

if (!result)

{

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

return;

}

//cdt.propagateDistance(CHAMFER\_345, &pDlg);

cdt.propagateDistance(&pDlg);

//convert the grid to a cloud

dPointCloud\* gridCloud = new dPointCloud(entity->getName() + QString(".distance\_grid(%1)").arg(steps));

{

unsigned pointCount = steps\*steps\*steps;

if (!gridCloud->reserve(pointCount))

{

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

delete gridCloud;

return;

}

dScalarField\* sf = new dScalarField("DT values");

if (!sf->reserveSafe(pointCount))

{

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

delete gridCloud;

sf->release();

return;

}

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

{

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

{

for (unsigned k = 0; k < steps; ++k)

{

ScalarType d = std::sqrt(static\_cast<ScalarType>(cdt.getValue(i, j, k))) \* cellDim;

if (!filterRange || (d >= range[0] && d <= range[1]))

{

gridCloud->addPoint(minCorner + CCVector3(i + 0.5, j + 0.5, k + 0.5) \* cellDim);

sf->addElement(d);

}

}

}

}

sf->computeMinAndMax();

int sfIdx = gridCloud->addScalarField(sf);

if (gridCloud->size() == 0)

{

dLog::Warning(QString("[DistanceMap] Cloud '%1': no point falls inside the specified range").arg(entity->getName()));

delete gridCloud;

gridCloud = nullptr;

}

else

{

gridCloud->setCurrentDisplayedScalarField(sfIdx);

gridCloud->showSF(true);

gridCloud->setDisplay(entity->getDisplay());

gridCloud->shrinkToFit();

entity->prepareDisplayForRefresh();

addToDB(gridCloud);

}

}

}

refreshAll();

}

void MainWindow::doActionComputeDistToBestFitQuadric3D()

{

bool ok = true;

int steps = QInputDialog::getInt(this,"Distance to best fit quadric (3D)","Steps (per dim.)",50,10,10000,10,&ok);

if (!ok)

return;

for ( dHObject \*entity : getSelectedEntities() )

{

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

{

dGenericPointCloud\* cloud = dHObjectCaster::ToGenericPointCloud(entity);

CCLib::Neighbourhood Yk(cloud);

double Q[10];

if (Yk.compute3DQuadric(Q))

{

const double& a = Q[0];

const double& b = Q[1];

const double& c = Q[2];

const double& e = Q[3];

const double& f = Q[4];

const double& g = Q[5];

const double& l = Q[6];

const double& m = Q[7];

const double& n = Q[8];

const double& d = Q[9];

//gravity center

const CCVector3\* G = Yk.getGravityCenter();

if (!G)

{

dConsole::Warning(QString("Failed to get the center of gravity of cloud '%1'!").arg(cloud->getName()));

continue;

}

const dBBox bbox = cloud->getOwnBB();

PointCoordinateType maxDim = bbox.getMaxBoxDim();

CCVector3 C = bbox.getCenter();

//Sample points on a cube and compute for each of them the distance to the Quadric

dPointCloud\* newCloud = new dPointCloud();

if (!newCloud->reserve(steps\*steps\*steps))

{

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

}

const char defaultSFName[] = "Dist. to 3D quadric";

int sfIdx = newCloud->getScalarFieldIndexByName(defaultSFName);

if (sfIdx < 0)

sfIdx = newCloud->addScalarField(defaultSFName);

if (sfIdx < 0)

{

dConsole::Error("Couldn't allocate a new scalar field for computing distances! Try to free some memory ...");

delete newCloud;

continue;

}

dScalarField\* sf = static\_cast<dScalarField\*>(newCloud->getScalarField(sfIdx));

assert(sf);

//FILE\* fp = fopen("doActionComputeQuadric3D\_trace.txt","wt");

for (int x = 0; x < steps; ++x)

{

CCVector3 P;

P.x = C.x + maxDim \* (static\_cast<PointCoordinateType>(x) / static\_cast<PointCoordinateType>(steps - 1) - PC\_ONE / 2);

for (int y = 0; y < steps; ++y)

{

P.y = C.y + maxDim \* (static\_cast<PointCoordinateType>(y) / static\_cast<PointCoordinateType>(steps - 1) - PC\_ONE / 2);

for (int z = 0; z < steps; ++z)

{

P.z = C.z + maxDim \* (static\_cast<PointCoordinateType>(z) / static\_cast<PointCoordinateType>(steps - 1) - PC\_ONE / 2);

newCloud->addPoint(P);

//compute distance to quadric

CCVector3 Pc = P-\*G;

ScalarType dist = static\_cast<ScalarType>( a\*Pc.x\*Pc.x + b\*Pc.y\*Pc.y + c\*Pc.z\*Pc.z

+ e\*Pc.x\*Pc.y + f\*Pc.y\*Pc.z + g\*Pc.x\*Pc.z

+ l\*Pc.x + m\*Pc.y + n\*Pc.z + d);

sf->addElement(dist);

//fprintf(fp,"%f %f %f %f\n",Pc.x,Pc.y,Pc.z,dist);

}

}

}

//fclose(fp);

if (sf)

{

sf->computeMinAndMax();

newCloud->setCurrentDisplayedScalarField(sfIdx);

newCloud->showSF(true);

}

newCloud->setName("Distance map to 3D quadric");

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

newCloud->prepareDisplayForRefresh();

addToDB(newCloud);

}

else

{

dConsole::Warning(QString("Failed to compute 3D quadric on cloud '%1'").arg(cloud->getName()));

}

}

}

refreshAll();

}

void MainWindow::doActionComputeCPS()

{

if (m\_selectedEntities.size() != 2)

{

dConsole::Error("Select 2 point clouds!");

return;

}

if (!m\_selectedEntities[0]->isKindOf(CC\_TYPES::POINT\_CLOUD) ||

!m\_selectedEntities[1]->isKindOf(CC\_TYPES::POINT\_CLOUD))

{

dConsole::Error("Select 2 point clouds!");

return;

}

dOrderChoiceDlg dlg( m\_selectedEntities[0], "Compared",

m\_selectedEntities[1], "Reference",

this );

if (!dlg.exec())

return;

dGenericPointCloud\* compCloud = dHObjectCaster::ToGenericPointCloud(dlg.getFirstEntity());

dGenericPointCloud\* srcCloud = dHObjectCaster::ToGenericPointCloud(dlg.getSecondEntity());

if (!compCloud->isA(CC\_TYPES::POINT\_CLOUD)) //TODO

{

dConsole::Error("Compared cloud must be a real point cloud!");

return;

}

dPointCloud\* cmpPC = static\_cast<dPointCloud\*>(compCloud);

static const char DEFAULT\_CPS\_TEMP\_SF\_NAME[] = "CPS temporary";

int sfIdx = cmpPC->getScalarFieldIndexByName(DEFAULT\_CPS\_TEMP\_SF\_NAME);

if (tempGroup.getChildrenNumber() != 0)

{

dBBox box = tempGroup.getDisplayBB\_recursive(false, win);

if (!box.isValid())

{

dLog::Warning("Selected entities have no valid bounding-box!");

}

else

{

if ( win != nullptr )

{

win->updateConstellationCenterAndZoom(&box);

}

}

}

refreshAll();

}

void MainWindow::setGlobalZoom()

{

dGLWindow\* win = getActiveGLWindow();

if (win)

win->zoomGlobal();

}

void MainWindow::setPivotAlwaysOn()

{

dGLWindow\* win = getActiveGLWindow();

if (win)

{

win->setPivotVisibility(dGLWindow::PIVOT\_ALWAYS\_SHOW);

win->redraw();

//update pop-up menu 'top' icon

if (m\_pivotVisibilityPopupButton)

m\_pivotVisibilityPopupButton->setIcon(m\_UI->actionSetPivotAlwaysOn->icon());

}

}

void MainWindow::setPivotRotationOnly()

{

dGLWindow\* win = getActiveGLWindow();

if (win)

{

win->setPivotVisibility(dGLWindow::PIVOT\_SHOW\_ON\_MOVE);

win->redraw();

//update pop-up menu 'top' icon

if (m\_pivotVisibilityPopupButton)

m\_pivotVisibilityPopupButton->setIcon(m\_UI->actionSetPivotRotationOnly->icon());

}

}

void MainWindow::setPivotOff()

{

dGLWindow\* win = getActiveGLWindow();

if (win)

{

win->setPivotVisibility(dGLWindow::PIVOT\_HIDE);

win->redraw();

//update pop-up menu 'top' icon

if (m\_pivotVisibilityPopupButton)

m\_pivotVisibilityPopupButton->setIcon(m\_UI->actionSetPivotOff->icon());

}

}

void MainWindow::setOrthoView(dGLWindow\* win)

{

if (win)

{

if (!checkStereoMode(win))

{

return;

}

win->setPerspectiveState(false, true);

win->redraw();

//update pop-up menu 'top' icon

if (m\_viewModePopupButton)

m\_viewModePopupButton->setIcon(m\_UI->actionSetOrthoView->icon());

if (m\_pivotVisibilityPopupButton)

m\_pivotVisibilityPopupButton->setEnabled(true);

}

}

void MainWindow::setCenteredPerspectiveView(dGLWindow\* win, bool autoRedraw/\*=true\*/)

{

if (win)

{

win->setPerspectiveState(true, true);

if (autoRedraw)

win->redraw();

//update pop-up menu 'top' icon

if (m\_viewModePopupButton)

m\_viewModePopupButton->setIcon(m\_UI->actionSetCenteredPerspectiveView->icon());

if (m\_pivotVisibilityPopupButton)

m\_pivotVisibilityPopupButton->setEnabled(true);

}

}

void MainWindow::setViewerPerspectiveView(dGLWindow\* win)

{

if (win)

{

win->setPerspectiveState(true,false);

win->redraw();

//update pop-up menu 'top' icon

if (m\_viewModePopupButton)

m\_viewModePopupButton->setIcon(m\_UI->actionSetViewerPerspectiveView->icon());

if (m\_pivotVisibilityPopupButton)

m\_pivotVisibilityPopupButton->setEnabled(false);

}

}

void MainWindow::enablePickingOperation(dGLWindow\* win, QString message)

{

if (!win)

{

assert(false);

return;

}

assert(m\_pickingHub);

if (!m\_pickingHub->addListener(this))

{

dLog::Error("Can't start the picking mechanism (another tool is already using it)");

return;

}

//specific case: we prevent the 'point-pair based alignment' tool to process the picked point!

//if (m\_pprDlg)

// m\_pprDlg->pause(true);

s\_pickingWindow = win;

win->displayNewMessage(message, dGLWindow::LOWER\_LEFT\_MESSAGE, true, 24 \* 3600);

win->redraw(true, false);

freezeUI(true);

}

void MainWindow::cancelPreviousPickingOperation(bool aborted)

{

if (!s\_pickingWindow)

return;

switch(s\_currentPickingOperation)

{

case PICKING\_ROTATION\_CENTER:

//nothing to do

break;

case PICKING\_LEVEL\_POINTS:

if (s\_levelMarkersCloud)

{

s\_pickingWindow->removeFromOwnDB(s\_levelMarkersCloud);

delete s\_levelMarkersCloud;

s\_levelMarkersCloud = nullptr;

}

break;

default:

assert(false);

break;

}

if (aborted)

{

s\_pickingWindow->displayNewMessage(QString(), dGLWindow::LOWER\_LEFT\_MESSAGE); //clear previous messages

s\_pickingWindow->displayNewMessage("Picking operation aborted", dGLWindow::LOWER\_LEFT\_MESSAGE);

}

s\_pickingWindow->redraw(false);

//specific case: we allow the 'point-pair based alignment' tool to process the picked point!

if (m\_pprDlg)

m\_pprDlg->pause(false);

freezeUI(false);

m\_pickingHub->removeListener(this);

s\_pickingWindow = nullptr;

s\_currentPickingOperation = NO\_PICKING\_OPERATION;

}

void MainWindow::onItemPicked(const PickedItem& pi)

{

if (!s\_pickingWindow || !m\_pickingHub)

{

return;

}

if (!pi.entity)

{

return;

}

if (m\_pickingHub->activeWindow() != s\_pickingWindow)

{

dLog::Warning("The point picked was picked in the wrong window");

return;

}

CCVector3 pickedPoint = pi.P3D;

switch(s\_currentPickingOperation)

{

case PICKING\_LEVEL\_POINTS:

{

//we only adept points picked on the right entity!

//if (obj != s\_levelEntity)

//{

// dLog::Warning(QString("[Level] Only points picked on '%1' are considered!").arg(s\_levelEntity->getName()));

// return;

//}

if (!s\_levelMarkersCloud)

{

assert(false);

cancelPreviousPickingOperation(true);

}

for (unsigned i = 0; i < s\_levelMarkersCloud->size(); ++i)

{

const CCVector3\* P = s\_levelMarkersCloud->getPoint(i);

if ((pickedPoint - \*P).norm() < 1.0e-6)

{

dLog::Warning("[Level] Point is too close from the others!");

return;

}

}

//add the corresponding marker

s\_levelMarkersCloud->addPoint(pickedPoint);

unsigned markerCount = s\_levelMarkersCloud->size();

d2DLabel\* label = new d2DLabel();

label->addPickedPoint(s\_levelMarkersCloud, markerCount - 1);

label->setName(QString("P#%1").arg(markerCount));

label->setDisplayedIn2D(false);

label->setDisplay(s\_pickingWindow);

label->setVisible(true);

s\_levelMarkersCloud->addChild(label);

s\_pickingWindow->redraw();

if (markerCount == 3)

{

//we have enough points!

const CCVector3\* A = s\_levelMarkersCloud->getPoint(0);

const CCVector3\* B = s\_levelMarkersCloud->getPoint(1);

const CCVector3\* C = s\_levelMarkersCloud->getPoint(2);

CCVector3 X = \*B - \*A;

CCVector3 Y = \*C - \*A;

CCVector3 Z = X.cross(Y);

//we choose 'Z' so that it points 'upward' relatively to the camera (assuming the user will be looking from the top)

CCVector3d viewDir = s\_pickingWindow->getCurrentViewDir();

if (CCVector3d::fromArray(Z.u).dot(viewDir) > 0)

{

Z = -Z;

}

Y = Z.cross(X);

X.normalize();

Y.normalize();

Z.normalize();

dGLMatrixd trans;

double\* mat = trans.data();

mat[0] = X.x; mat[4] = X.y; mat[8] = X.z; mat[12] = 0;

mat[1] = Y.x; mat[5] = Y.y; mat[9] = Y.z; mat[13] = 0;

mat[2] = Z.x; mat[6] = Z.y; mat[10] = Z.z; mat[14] = 0;

mat[3] = 0 ; mat[7] = 0 ; mat[11] = 0 ; mat[15] = 1;

CCVector3d T = -CCVector3d::fromArray(A->u);

trans.apply(T);

T += CCVector3d::fromArray(A->u);

trans.setTranslation(T);

assert(haveOneSelection() && m\_selectedEntities.front() == s\_levelEntity);

applyTransformation(trans);

//clear message

s\_pickingWindow->displayNewMessage(QString(), dGLWindow::LOWER\_LEFT\_MESSAGE, false); //clear previous message

s\_pickingWindow->setView(CC\_TOP\_VIEW);

}

else

{

//we need more points!

return;

}

}

//we use the next 'case' entry (PICKING\_ROTATION\_CENTER) to redefine the rotation center as well!

assert(s\_levelMarkersCloud && s\_levelMarkersCloud->size() != 0);

pickedPoint = \*s\_levelMarkersCloud->getPoint(0);

//break;

case PICKING\_ROTATION\_CENTER:

{

CCVector3d newPivot = CCVector3d::fromArray(pickedPoint.u);

//specific case: transformation tool is enabled

if (m\_transTool && m\_transTool->started())

{

m\_transTool->setRotationCenter(newPivot);

const unsigned& precision = s\_pickingWindow->getDisplayParameters().displayedNumPrecision;

s\_pickingWindow->displayNewMessage(QString(), dGLWindow::LOWER\_LEFT\_MESSAGE, false); //clear previous message

s\_pickingWindow->displayNewMessage(QString("Point (%1 ; %2 ; %3) set as rotation center for interactive transformation")

.arg(pickedPoint.x, 0, 'f', precision)

.arg(pickedPoint.y, 0, 'f', precision)

.arg(pickedPoint.z, 0, 'f', precision),

dGLWindow::LOWER\_LEFT\_MESSAGE, true);

}

else

{

const dViewportParameters& params = s\_pickingWindow->getViewportParameters();

if (!params.perspectiveView || params.objectCenteredView)

{

//apply current GL transformation (if any)

pi.entity->getGLTransformation().apply(newPivot);

s\_pickingWindow->setPivotPoint(newPivot, true, true);

}

}

//s\_pickingWindow->redraw(); //already called by 'cancelPreviousPickingOperation' (see below)

}

break;

default:

assert(false);

break;

}

cancelPreviousPickingOperation(false);

}

void MainWindow::doPickRotationCenter()

{

//picking operation already in progress

if (s\_pickingWindow)

{

if (s\_currentPickingOperation == PICKING\_ROTATION\_CENTER)

{

cancelPreviousPickingOperation(true);

}

else

{

dConsole::Error("Stop the other picking operation first!");

}

return;

}

dGLWindow\* win = getActiveGLWindow();

if (!win)

{

dConsole::Error("No active 3D view!");

return;

}

bool objectCentered = true;

bool perspectiveEnabled = win->getPerspectiveState(objectCentered);

if (perspectiveEnabled && !objectCentered)

{

dLog::Error("Perspective mode is viewer-centered: can't use a point as rotation center!");

return;

}

s\_currentPickingOperation = PICKING\_ROTATION\_CENTER;

enablePickingOperation(win, "Pick a point to be used as rotation center (click on icon again to cancel)");

}

void MainWindow::setView( CC\_VIEW\_ORIENTATION view )

{

dGLWindow\* win = getActiveGLWindow();

if (win)

{

win->setView(view);

}

}

void MainWindow::doActionCreateCloudFromEntCenters()

{

size\_t selNum = getSelectedEntities().size();

dPointCloud\* centers = new dPointCloud("centers");

if (!centers->reserve(static\_cast<unsigned>(selNum)))

{

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

delete centers;

centers = nullptr;

return;

}

//look for clouds

{

for ( dHObject \*entity : getSelectedEntities() )

{

dPointCloud\* cloud = dHObjectCaster::ToPointCloud(entity);

if (cloud == nullptr)

{

continue;

}

centers->addPoint(cloud->getOwnBB().getCenter());

//we display the cloud in the same window as the first (selected) cloud we encounter

if (!centers->getDisplay())

{

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

}

}

}

if (centers->size() == 0)

{

dLog::Error("No cloud in selection?!");

delete centers;

centers = nullptr;

}

else

{

centers->resize(centers->size());

centers->setPointSize(10);

centers->setVisible(true);

addToDB(centers);

}

}

bool MainWindow::checkStereoMode(dGLWindow\* win)

{

assert(win);

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

{

dGLWindow::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()

{

dGLWindow\* win = getActiveGLWindow();

if (win)

{

const dViewportParameters& 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()

{

dGLWindow\* win = getActiveGLWindow();

if (win)

{

const dViewportParameters& 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()

{

dGLWindow\* win = getActiveGLWindow();

if (win)

{

bool wasLocked = win->isRotationAxisLocked();

bool isLocked = !wasLocked;

static CCVector3d s\_lastAxis(0.0, 0.0, 1.0);

if (isLocked)

{

dAskThreeDoubleValuesDlg 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]"), dGLWindow::UPPER\_CENTER\_MESSAGE, false, 24 \* 3600, dGLWindow::ROTAION\_LOCK\_MESSAGE);

}

else

{

win->displayNewMessage(QString(), dGLWindow::UPPER\_CENTER\_MESSAGE, false, 0, dGLWindow::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\_dRoot)

{

dHObject::Container selectedEntities;

m\_dRoot->getSelectedEntities(selectedEntities);

if (selectedEntities.size() == 1)

{

dHObject\* ent = selectedEntities.front();

dGBLSensor\* sensor = nullptr;

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

{

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

}

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

{

dHObject::Container sensors;

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

if (sensors.size() >= 1)

{

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

}

}

if (sensor)

{

sensor->applyViewport();

return;

}

}

}

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

dGLWindow\* win = getActiveGLWindow();

if (win)

{

win->setBubbleViewMode(true);

win->redraw(false);

}

}

void MainWindow::removeFromDB(dHObject\* 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\_dRoot)

m\_dRoot->removeElement(obj);

}

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

{

if (obj && m\_dRoot)

{

if (selected)

m\_dRoot->selectEntity(obj);

else

m\_dRoot->unselectEntity(obj);

}

}

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();

}

}

dColorScalesManager\* MainWindow::getColorScalesManager()

{

return dColorScalesManager::GetUniqueInstance();

}

void MainWindow::closeAll()

{

if (!m\_dRoot)

{

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\_dRoot->unloadAll();

redrawAll(false);

}

void MainWindow::doActionLoadFile()

{

//persistent settings

QSettings settings;

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

QString currentPath = settings.value(dPS::CurrentPath(), dFileUtils::defaultDocPath()).toString();

QString currentOpenDlgFilter = settings.value(dPS::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,

CCFileDialogOptions());

if (selectedFiles.isEmpty())

return;

//save last loading parameters

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

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

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

settings.endGroup();

//load files

addToDB(selectedFiles, currentOpenDlgFilter);

}

//Helper: check for a filename validity

static bool IsValidFileName(QString filename)

{

#ifdef CC\_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;

dHObject clouds("clouds");

dHObject meshes("meshes");

dHObject images("images");

dHObject polylines("polylines");

dHObject other("other");

dHObject otherSerializable("serializable");

dHObject::Container entitiesToDispatch;

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

dHObject entitiesToSave;

while (!entitiesToDispatch.empty())

{

dHObject\* child = entitiesToDispatch.back();

entitiesToDispatch.pop\_back();

if (child->isA(CC\_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

dHObject\* dest = nullptr;

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

dest = &clouds;

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

dest = &meshes;

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

dest = &images;

else if (child->isKindOf(CC\_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, dHObject::DP\_NONE);

entitiesToSave.addChild(child, dHObject::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)

{

dConsole::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(CC\_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(CC\_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(CC\_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(CC\_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

{

CC\_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)

{

dHObject\* 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(dPS::SaveFile());

//default filter

QString selectedFilter = fileFilters.first();

if (hasCloud)

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

else if (hasMesh)

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

else if (hasImages)

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

else if (hasPolylines)

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

//default output path (+ filename)

QString currentPath = settings.value(dPS::CurrentPath(), dFileUtils::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(CC\_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))

{

dLog::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,

CCFileDialogOptions());

if (selectedFilename.isEmpty())

{

//process cancelled by the user

return;

}

//ignored items

if (hasOther)

{

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

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

{

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

}

}

CC\_FILE\_ERROR result = CC\_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

dHObject tempContainer;

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

if (tempContainer.getChildrenNumber())

{

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

}

}

}

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

{

//ignored items

//if (hasSerializable)

//{

// if (!hasOther)

// dConsole::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)

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

//}

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

selectedFilename,

parameters,

selectedFilter);

if (result == CC\_FERR\_NO\_ERROR && m\_dRoot)

{

m\_dRoot->unselectAllEntities();

}

}

//update default filters

if (hasCloud)

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

if (hasMesh)

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

if (hasImages)

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

if (hasPolylines)

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

//we update current file path

currentPath = QFileInfo(selectedFilename).absolutePath();

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

settings.endGroup();

}

void MainWindow::on3DViewActivated(QMdiSubWindow\* mdiWin)

{

dGLWindow\* 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(dGLWindow\* 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(dGLWindow\* win)

{

if (!m\_pivotVisibilityPopupButton)

return;

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

if (win)

{

QAction\* visibilityAction = nullptr;

switch(win->getPivotVisibility())

{

case dGLWindow::PIVOT\_HIDE:

visibilityAction = m\_UI->actionSetPivotOff;

break;

case dGLWindow::PIVOT\_SHOW\_ON\_MOVE:

visibilityAction = m\_UI->actionSetPivotRotationOnly;

break;

case dGLWindow::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()

{

dGLWindow\* active3DView = getActiveGLWindow();

bool hasMdiChild = (active3DView != nullptr);

int mdiChildCount = getGLWindowCount();

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

bool hasSelectedEntities = (m\_dRoot && m\_dRoot->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 )

{

dGLWindow \*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();

}

void MainWindow::updatePropertiesView()

{

if (m\_dRoot)

{

m\_dRoot->updatePropertiesView();

}

}

void MainWindow::updateUIWithSelection()

{

dbTreeSelectionInfo selInfo;

m\_selectedEntities.clear();

if (m\_dRoot)

{

m\_dRoot->getSelectedEntities(m\_selectedEntities, CC\_TYPES::OBJECT, &selInfo);

}

enableUIItems(selInfo);

}

void MainWindow::enableAll()

{

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

{

window->setEnabled( true );

}

}

void MainWindow::disableAll()

{

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

{

window->setEnabled( false );

}

}

void MainWindow::disableAllBut(dGLWindow\* win)

{

//we disable all other windows

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

{

if (GLWindowFromWidget(window->widget()) != win)

{

window->setEnabled(false);

}

}

}

void MainWindow::doActionResampleWithOctree()

{

bool ok;

int pointCount = QInputDialog::getInt(this, "Resample with octree", "Points (approx.)", 1000000, 1, INT\_MAX, 100000, &ok);

if (!ok)

return;

dProgressDialog pDlg(false, this);

pDlg.setAutoClose(false);

assert(pointCount > 0);

unsigned aimedPoints = static\_cast<unsigned>(pointCount);

bool errors = false;

for (dHObject \*entity : getSelectedEntities())

{

dPointCloud\* cloud = nullptr;

/\*if (ent->isKindOf(CC\_TYPES::MESH)) //TODO

cloud = dHObjectCaster::ToGenericMesh(ent)->getAssociatedCloud();

else \*/

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

{

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

}

if (cloud)

{

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

if (!octree)

{

octree = cloud->computeOctree(&pDlg);

if (!octree)

{

dConsole::Error(QString("Could not compute octree for cloud '%1'").arg(cloud->getName()));

continue;

}

}

cloud->setEnabled(false);

QElapsedTimer eTimer;

eTimer.start();

CCLib::GenericIndexedCloud\* result = CCLib::CloudSamplingTools::resampleCloudWithOctree

(

cloud,

aimedPoints,

CCLib::CloudSamplingTools::CELL\_GRAVITY\_CENTER,

&pDlg,

octree.data()

);

if (result)

{

dConsole::Print("[ResampleWithOctree] Timing: %3.2f s.", eTimer.elapsed() / 1.0e3);

dPointCloud\* newCloud = dPointCloud::From(result, cloud);

delete result;

result = nullptr;

if (newCloud)

{

addToDB(newCloud);

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

newCloud->prepareDisplayForRefresh();

}

else

{

errors = true;

}

}

}

}

if (errors)

dLog::Error("[ResampleWithOctree] Errors odurred during the process! Result may be incomplete!");

refreshAll();

}

void MainWindow::doActionSORFilter()

{

dSORFilterDlg sorDlg(this);

//set semi-persistent/dynamic parameters

static int s\_sorFilterKnn = 6;

static double s\_sorFilterNSigma = 1.0;

sorDlg.knnSpinBox->setValue(s\_sorFilterKnn);

sorDlg.nSigmaDoubleSpinBox->setValue(s\_sorFilterNSigma);

if (!sorDlg.exec())

return;

//update semi-persistent/dynamic parameters

s\_sorFilterKnn = sorDlg.knnSpinBox->value();

s\_sorFilterNSigma = sorDlg.nSigmaDoubleSpinBox->value();

dProgressDialog pDlg(true, this);

pDlg.setAutoClose(false);

bool firstCloud = true;

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

for (dHObject \*entity : selectedEntities)

{

//specific test for locked vertices

bool lockedVertices;

dPointCloud\* cloud = dHObjectCaster::ToPointCloud(entity, &lockedVertices);

if (cloud && lockedVertices)

{

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

continue;

}

//computation

CCLib::ReferenceCloud\* selection = CCLib::CloudSamplingTools::sorFilter(cloud,

s\_sorFilterKnn,

s\_sorFilterNSigma,

nullptr,

&pDlg);

if (selection && cloud)

{

if (selection->size() == cloud->size())

{

dLog::Warning(QString("[doActionSORFilter] No points were removed from cloud '%1'").arg(cloud->getName()));

}

else

{

dPointCloud\* cleanCloud = cloud->partialClone(selection);

if (cleanCloud)

{

cleanCloud->setName(cloud->getName() + QString(".SOR"));

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

if (cloud->getParent())

cloud->getParent()->addChild(cleanCloud);

addToDB(cleanCloud);

cloud->setEnabled(false);

if (firstCloud)

{

dConsole::Warning("Previously selected entities (sources) have been hidden!");

firstCloud = false;

m\_dRoot->selectEntity(cleanCloud, true);

}

}

else

{

dConsole::Warning(QString("[doActionSORFilter] Not enough memory to create a clean version of cloud '%1'!").arg(cloud->getName()));

}

}

delete selection;

selection = nullptr;

}

else

{

//no points fall inside selection!

if (cloud != nullptr)

{

dConsole::Warning(QString("[doActionSORFilter] Failed to apply the noise filter to cloud '%1'! (not enough memory?)").arg(cloud->getName()));

}

else

{

dConsole::Warning("[doActionSORFilter] Trying to apply the noise filter to null cloud");

}

}

}

refreshAll();

updateUI();

}