



•Hagenberg •Linz •Steyr •Wels

Dokumentation

Pendulum Painter

Datenaufbereitung / Visualisierung	
Thema	2D und 3D Simulation eines Farbpendels
Betreuer	DI(FH) Dr. Christoph Heinzl
Name / Matrikelnummer	Julian Kastenhuber / S2010566006 Patrick Holzer / S2010566005
Masterstudiengang	Entwicklungsingenieur*in Maschinenbau
Abgabe	08.03.2022

Inhaltsverzeichnis

1	Einführung.....	3
1.1	Motivation.....	3
1.2	Entwurf.....	3
2	Vorbereitung.....	5
2.1	Konfiguration von Boost	5
2.2	Konfiguration und bauen von VTK in Qt	5
3	Klassen	7
3.1	SphericalPendulum	7
3.2	PendulumPainter.....	10
4	GUI	16
4.1	Aufbau der Benutzeroberfläche	16
4.2	Benutzereingabe und Interaktion.....	18
4.3	Animation	19
5	Zusammenfassung.....	21
6	Abbildungsverzeichnis.....	23
7	Tabellenverzeichnis.....	24

1 Einführung

1.1 Motivation

Beim sogenannten „Pendulum Painting“ führt ein an einer Schnur befestigter Farbkübel mit einem Loch eine Pendelbewegung aus. Dadurch können verschiedenste Muster, je nach Auslenkung und Geschwindigkeit entstehen. Das führt zu kreativen Bildern, wie in Abbildung 1-1 zu sehen ist.

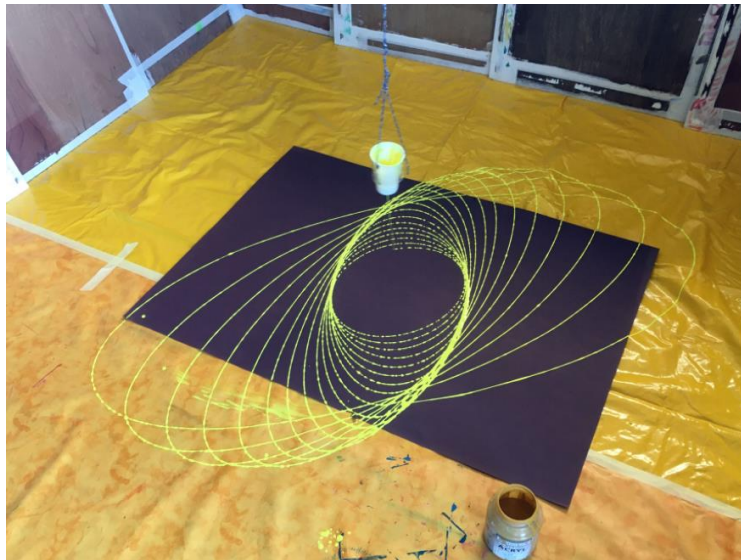


Abbildung 1-1 „Pendulum Painting“

Das Prinzip dahinter ist auf das Newtonsche Gesetz zurückzuführen, das wir in der Dynamik kennengelernt haben und mittels Lagrangescher Mechanik werden die Bewegungen hergeleitet.

Daraus ergibt sich eine Aufgabe, wo Grundelemente des Maschinenbaustudiums mit der visuellen Programmierung in C++ verknüpft werden.

1.2 Entwurf

Bevor mit der Programmierung begonnen wird, erfolgt ein Paper Prototyping. Dabei wird die Grafische Benutzeroberfläche skizziert und die darin eingebetteten Funktionen definiert. Dieser Entwurf soll dann als Anhaltspunkt zur Erstellung der GUI, Programmierung der Funktionen und Aufteilung in unterschiedlichen Klassen dienen. Beim Entwurf wurden folgende Funktionen berücksichtigt:

- Simulation in 2D und 3D Ansicht
- Steuerung: z.B Simulationsgeschwindigkeit, Start/Stop
- Eingabe und Visualisierung der Startwerte
- Änderung der Farben einzelner Ansichten
- Hintergrundmusik

Die nachfolgende Abbildung 1-2 zeigt den erstellten Entwurf der GUI.

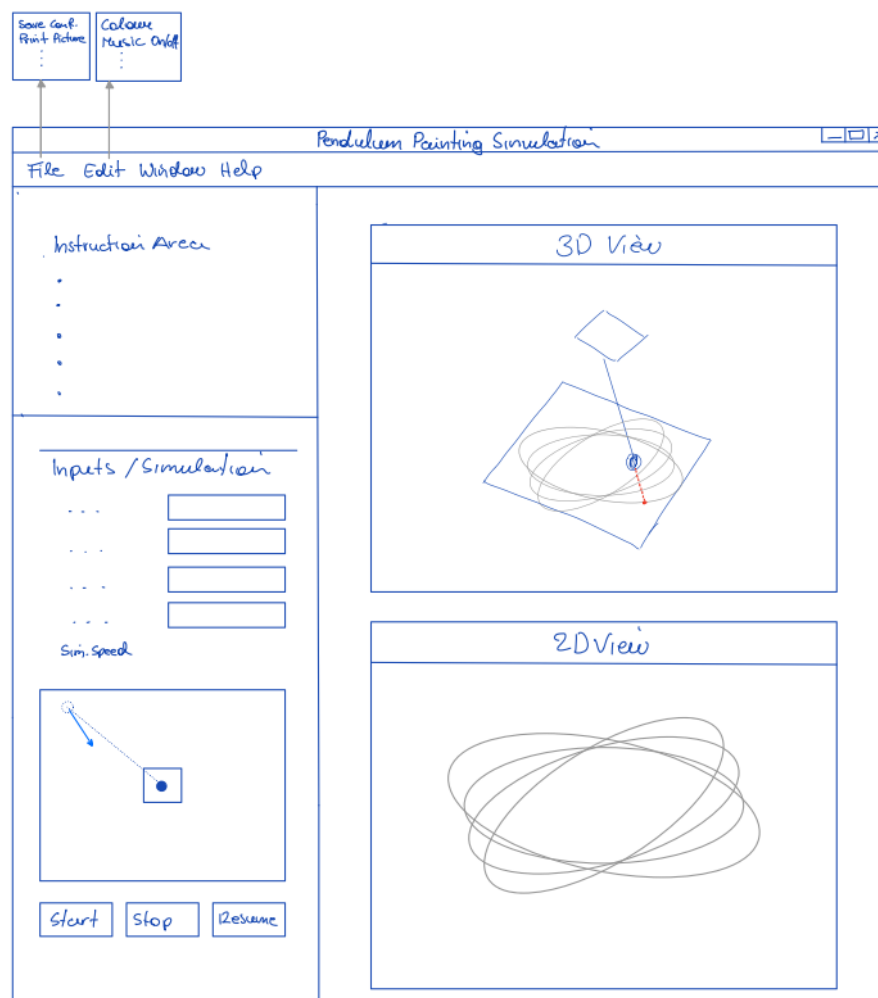


Abbildung 1-2 Entwurf der grafischen Oberfläche und den darin enthaltenen Funktionen.

2 Vorbereitung

2.1 Konfiguration von Boost

1. Laden Sie Boost_1.78.0.zip Datei herunter
(https://www.boost.org/users/history/version_1_78_0.html)
2. Entpacken der zip Datei nach z.B "C:\boost_1_78_0". Beliebiger Pfad wählbar, aber nachfolgende Anweisungen beziehen sich dann immer auf diesen Pfad.
3. Öffner der Kommandozeile in diesem Ordner (Rechtsklick und öffnen des Windows Terminals)
 - In der Kommandozeile "bootstrap.bat" eingeben und dann wird boost vorbereitet für den Build.
 - Ist die Vorbereitung abgeschlossen ".b2" eingeben und jetzt wird boost gebildet (kann paar Minuten dauern).
4. Build abgeschlossen, wenn steht "The Boost C++ Libraries were successfully built! "

2.2 Konfiguration und bauen von VTK in Qt

1. Starten sie CMake (cmake-gui)
2. PendulumPainter Projekt bauen
 - Feld - "Where is the source code": Klicken sie auf Browse Source und selektieren sie als Quellpfad den Ordner "C:\Tools\Src\project\PendulumPainter" bzw. entsprechenden anderen Pfad, wo Sie den Source Code abgelegt haben.
 - Feld - "Where to build the binaries": Geben sie den Pfad "C:\Tools\Pro\project\PendulumPainter" ein. In diesen Ordner wird das PendulumPainter Projekt gebaut bzw. entsprechenden anderen Pfad, wo Sie das PendulumPainter Projekt bauen möchten.
3. Konfigurieren sie ihren PendulumPainter build:
 - Drücke "Configure".
 - Aktivieren der Checkbox "Advanced" um alle Einträge zu sehen.
 - Wahrscheinlich erhalten Sie eine Fehlermeldung, dass VTK_DIR nicht gefunden werden konnte. Klicken Sie in diesem Fall auf die 3 Punkte im Feld VTK_DIR und setzen Sie die Variable VTK_DIR auf "C:/Tools/Pro/vtk", das ist der Ordner, wo VTK gebaut wurde. Ist VTK woanders gebildet, dann entsprechend diesen Pfad angeben.
 - Drücke nochmals "Configure" (müsste um vieles Schneller sein).
 - Wenn immer noch Fehler auftreten, sollten Ihnen die entsprechenden Meldungen einen Hinweis darauf geben, was zu tun ist, um sie zu beheben.
 - Wenn keine Fehler mehr auftreten und am unteren Ende des Protokolls "Configuring done" angezeigt wird, drücken Sie auf "Generate".

4. PendulumPainter ist nun bereit, Sie können auf „Open Project“ klicken. Stellen Sie sicher, dass mit der richtigen Visual Studio Version geöffnet wird, falls mehrere Versionen installiert sind.
5. Projektmappe erstellen
 - Die Debug-Konfiguration sollte in Visual Studio vorausgewählt sein. Über Erstellen >> Projektmappe erstellen kann der Build ausgelöst werden.
 - Wenn der Build ohne Fehler abgeschlossen wurde, sollte PendulumPainter betriebsbereit sein.
6. Legen Sie "PendulumPainter" als Startprojekt fest. Dazu klicken wir im Projektmappen-Explorer mit der rechten Maustaste auf unser "PendulumPainter"-Projekt (direkt unter ALL_BUILD) und wählen "Als Startprojekt festlegen". "PendulumPainter" sollte nun in fetter Schrift erscheinen. Dieses Projekt ist nun als Startprojekt ausgewählt.
7. Entsprechende Pfade definieren:

Da wir unserem Programm mitteilen müssen, in welcher Umgebung es laufen soll, d.h. auf welche Bibliotheken und *.dlls es Zugriff haben soll, müssen wir die Umgebung angeben. Dies kann durch einen weiteren Rechtsklick auf das Projekt "PendulumPainter" im Projektmappen-Explorer geschehen. Ganz unten finden Sie den Menüpunkt "Eigenschaften". Klicken Sie diesen an und es öffnet sich ein weiteres Fenster mit allen Projekteigenschaften.

 - Unter Debugging, um alle Debugging bezogenen Einstellungen dieses Projekts anzuzeigen. Suchen Sie nach dem Eintrag „Umgebung“ und geben Sie unsere Umgebung ein:
PATH=\$(PATH);C:\Tools\Pro\vtk\bin\Debug;C:\Qt\5.15.2\msvc2019_64\bin;
(Stellen Sie sicher, dass Sie das ";" am Ende nicht vergessen. Andernfalls werden die DLLs möglicherweise nicht gefunden.)
 - Um Zugriff auf die Boost Bibliothek zu haben, müssen wir noch unter C/C++ >> Allgemein >> Zusätzliche Includeverzeichnisse den Boost Pfad hinzufügen. Dazu rechts über das Dropdown bearbeiten anklicken und eine neue Zeile hinzufügen mit dem Pfad "C:\boost_1_78_0". Weiters muss unter Linker >> Allgemein >> Zusätzliche Bibliotheksverzeichnisse folgender Boost Pfad hinzugefügt werden. Dazu wieder rechts über das Dropdown bearbeiten anklicken und eine neue Zeile hinzufügen mit dem Pfad "C:\boost_1_78_0\stage\lib".
 - Bestätigen Sie die eingegebenen Einstellungen durch Drücken von ok.
8. PendulumPainter sollte jetzt startbereit sein. Drücken Sie die Wiedergabetaste neben dem lokalen Windows-Debugger im oberen Bereich von Visual Studio.

3 Klassen

Die Aufteilung in zwei Klassen basiert auf dem Gedanken ein paralleles Arbeiten am Code so einfach wie möglich gestalten zu können. Die eine Klasse übernimmt die mathematischen Operationen und die andere Klasse sämtliche GUI (Qt) sowie die zugehörigen 2D- und 3D-Fenster (VTK)

3.1 SphericalPendulum

3.1.1 Mathematische Beschreibung

In dieser Klasse werden alle mathematischen Algorithmen durchgeführt. Das Herzstück ist die Lösung des Differenzialgleichungssystems für das 3D-Pendel. Als Grundlage für die Aufstellung der Lagrangeschen Bewegungsgleichungen dient das ungedämpfte sphärische Pendel (Abbildung 3-1) aus dem Mechanik Vorlesungsskriptum III¹.

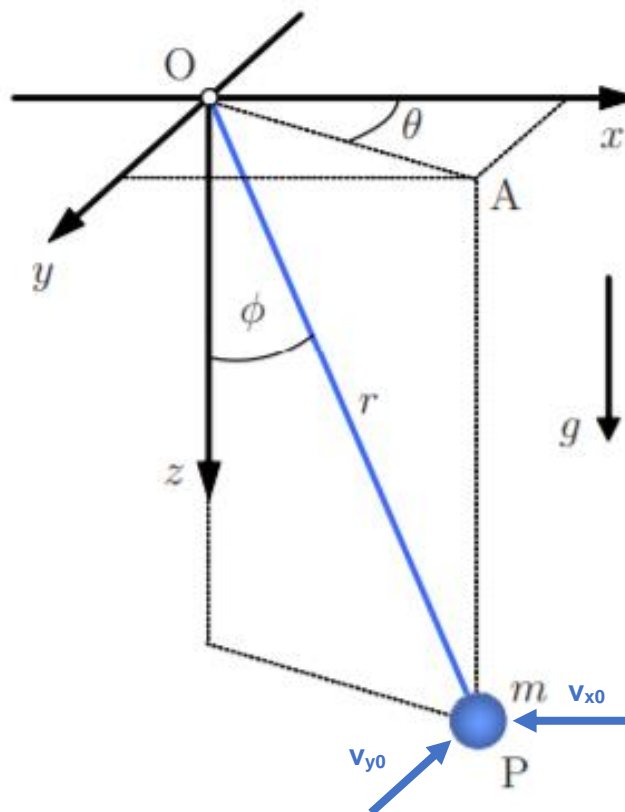


Abbildung 3-1 Sphärisches Pendel (auch Raum- oder Kugelpendel) (Steiner, 2015, p. 109)

Die dort aufgestellten Bewegungsgleichungen (5.41) und (5.42) werden um den Beitrag der Dämpfungskräfte erweitert.

$$Q_{d\phi} = -d_{\phi}\dot{\phi}$$

$$Q_{d\theta} = -d_{\theta}\dot{\theta}$$

¹ Steiner, W., 2015. Vorlesungsskriptum Technische Mechanik III. Wels: FH OÖ. S108-112

Daraus ergeben sich die Lagrangeschen Bewegungsgleichungen

$$\frac{d}{dt} \left(\frac{\partial L}{\partial \dot{\phi}} \right) - \frac{\partial L}{\partial \phi} = Q_{d\phi}$$

$$\frac{d}{dt} \left(\frac{\partial L}{\partial \dot{\theta}} \right) - \frac{\partial L}{\partial \theta} = Q_{d\theta}$$

Aus denen erhält man die Differenzialgleichungen 2.Ordnung, die nach dem Umformen folgendermaßen angeschrieben werden können

$$\ddot{\phi} = \dot{\theta}^2 \sin \phi \cos \phi - \frac{g}{r} \sin \phi - \frac{d_{\phi}}{m} \dot{\phi}$$

$$\ddot{\theta} = -\frac{2\dot{\phi}\dot{\theta} \cos \phi}{\sin \phi} - \frac{d_{\theta}}{m} \dot{\theta}$$

3.1.2 Programmierung

Um das Differenzialgleichungssystem (Ordinary Differential Equation System – ODE-System) mit odeint aus der boost Bibliothek lösen zu können, muss eine Orderreduktion auf ein ODE-System 1.Ordnung durchgeführt werden. Das erfolgt folgendermaßen:

$$\begin{aligned} x[0] &= \phi & x[1] &= \dot{\phi} = dxdt[0] & dxdt[1] &= \ddot{\phi} \\ x[2] &= \theta & x[3] &= \dot{\theta} = dxdt[2] & dxdt[3] &= \ddot{\theta} \end{aligned}$$

Das heißt, die ODEs 2. Ordnung werden in ODEs 1. Ordnung übergeführt. Dieses Differenzialgleichungssystem ist dann in der Funktion (siehe Abbildung 3-2) SphericalPendulum::defineODESystem(...) definiert.

```
// Introducing the ODEs as first order ODEs.
// Apply numerical integration design as the right hand side (RHS) of the equations.
void SphericalPendulum::defineODESystem(const stateType& x, stateType& dxdt, double t) {
    dxdt[0] = x[1];
    dxdt[1] = x[3] * x[3] * sin(x[0]) * cos(x[0]) - g / r * sin(x[0]) - d * x[1] / m;
    dxdt[2] = x[3];
    dxdt[3] = -((2 * x[3] * x[1] * cos(x[0])) / sin(x[0])) - d * x[3] / m;
}
```

Abbildung 3-2 Quellcode der Definition des ODE-Systems

Dieses ODE-System wird in die Funktion SphericalPendulum::integrateODE(...) übergeben. Die Startwerte, Berechnungszeit, Pendellänge sowie der Dämpfungsparameter werden über das GUI vom Benutzer eingegeben und mittels set-Funktionen in der Klasse definiert. Die Lösung des ODE-Systems wird mittels Vector- und Matrixreferenzen ausgegeben.


```
// Solving the ODE and outputs a result matrix and a time vector
// Matrix: each row is a step; a row contains in this order: phi, phi-dot, theta, theta-dot
void SphericalPendulum::integrateODE(matrix& matX, vector<double>& vecTime){
    // observe vectors
    vector<stateType> fMatX;
    vector<double> fVecTime;

    size_t steps = integrate([this](auto const& x, auto& dxdt, auto t) {this->defineODESystem(x, dxdt, t); },
        x0, timeSet[0], timeSet[1], timeSet[2], push_back_state_and_time(fMatX, fVecTime) );

    matX = fMatX;
    vecTime = fVecTime;
}
```

Abbildung 3-3 Quellcode vom Löser des ODE-Systems

Um die Lösung während der Integrationsschritte zu beobachten, wird ein entsprechender Beobachter² wie in Abbildung 3-4 bereitgestellt. Dieser speichert die Zwischenschritte in einem Container entsprechend der Argumentstruktur des ()-Operators. odeint ruft den Beobachter genau auf diese Weise auf, indem er den aktuellen Zustand und die Zeit angibt, und dieser Container wird an die Integrationsfunktion übergeben.

```
// Observe the solution during the integration steps (Standard from boost example: harmonic oscillator)
struct push_back_state_and_time {
    vector< stateType >& m_states;
    vector< double >& m_times;

    push_back_state_and_time(vector< stateType >& states, vector< double >& times)
        : m_states(states), m_times(times) { }

    void operator()(const stateType& x, double t) {
        m_states.push_back(x);
        m_times.push_back(t);
    }
};
```

Abbildung 3-4 Quellcode des ODE-Definition

Schlussendlich wird über die get-Funktion `SphericalPendulum::getMatVTK(...)` die benötigten Winkeländerungen pro Zeitschritt für die 3D-Pendelbewegungen sowie die x- und y-Koordinaten für die Darstellung der 2D-Kurve umgerechnet und in einer Matrix zurückgegeben.

Inkludierte Print Funktionen in der Klasse können aktiviert werden, um die berechneten Schritte im Kommandofenster auszugeben.

Die Klasse wird schlussendlich mit der privaten Funktion `runCalSphericalPendulum()` der `PendulumPainter` Klasse aufgerufen und ausgeführt, Details dazu im nachfolgenden Kapitel 3.2.

2

https://www.boost.org/doc/libs/1_54_0/libs/numeric/odeint/doc/html/boost_numeric_odeint/getting_started/short_example.html

3.2 PendulumPainter

3.2.1 Übersicht

Die Klasse PendulumPainter wird verwendet, um die berechneten Daten der Pendeldynamik, und das dazugehörige Muster zu visualisieren. Auch die Darstellung und Verarbeitung von Ein- und Ausgabedaten der Grafischen Benutzeroberfläche (GUI) ist Teil des Funktionsumfangs.

Die Klasse verwendet zwei Toolkits. Zum einen Qt welche für die Visualisierung der GUI verwendet wird und VTK mit dem die in der GUI laufenden Animationen erzeugt werden.

Qt verwendet das Signal-Slot-Konzept. Das bedeutet, dass Signale und Slots einen ereignisgesteuerten Programmfluss beziehungsweise eine ereignisgesteuerte Kommunikation zwischen Programmobjekten realisieren.

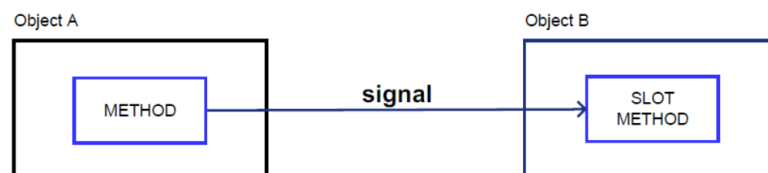


Abbildung 3-5 Signal-Slot-Konzept³

Nachfolgende Tabellen stellen eine Übersicht aller implementierten Funktionen sowie deren Aufgaben dar.

Private	SimUpdate3D()	Visualisiert ein Animations-Inkrement in 3D.
	SimUpdate2D()	Visualisiert ein Animations-Inkrement in 2D.
	initSim()	Liest die aktuellen Eingabeparameter aus dem GUI, resettet sämtliche Fortlaufvariablen der Simulationsschleife.
	init3DActors()	Erzeugt die Geometrien und definiert deren Positionen für die VTK-Pipeline. Es wird eine Baugruppe bestehend aus Kegel, Seile und Lagerpunkt erzeugt und initialisiert.
	runCalSphericalPendulum()	Erzeugt die für die Animation notwendigen Daten mit Hilfe der Berechnungsklasse.

Public	setCalData(matrix& matCal)	Eingabe der Daten aus der Berechnungsklasse für die weitere Verarbeitung.
	getDataGUI()	Ausgabe der aktuellen Werte im GUI.

Slots (Public)	initialize()	Wird beim Drücken des Initialisierungsbutton ausgeführt und Initialisiert das Programm.
	timerslot()	Schleife für die Simulation / Animation. Jeder Schleifendurchlauf updatet die Pendelposition sowie das gezeichnete Muster.
	pushButtonSim()	Notwendig für die Steuerung der Simulationsschleife timerslot()

³ Quelle: <http://ramontalaverasuarez.blogspot.com/2014/05/qt-signals-and-slots-connecting-and.html>

changeColor()	Ermöglicht das Wechseln sämtlicher Farben in der Grafischen Benutzeroberfläche.
changeColorDefault()	Setzt alle Farben auf den Default Wert.
saveImage()	Ermöglicht das Abspeichern der in der 2D Ansicht erstellten Zeichnung im .png Format.
getSliderValue()	Zum auslesen des Slider Wertes aus der GUI
slotExit()	Beim Drucken des Close Buttons wird die Anwendung beendet.

Tabelle 3-1 Beschreibung aller in der Klasse PendulumPainter vorhandenen Funktionen

3.2.2 Programmablauf

Folgende Ablaufdiagramm soll die Interaktionen der einzelnen Funktionen darstellen und einen groben Überblick über die Vorgehensweise der Programmierung geben.

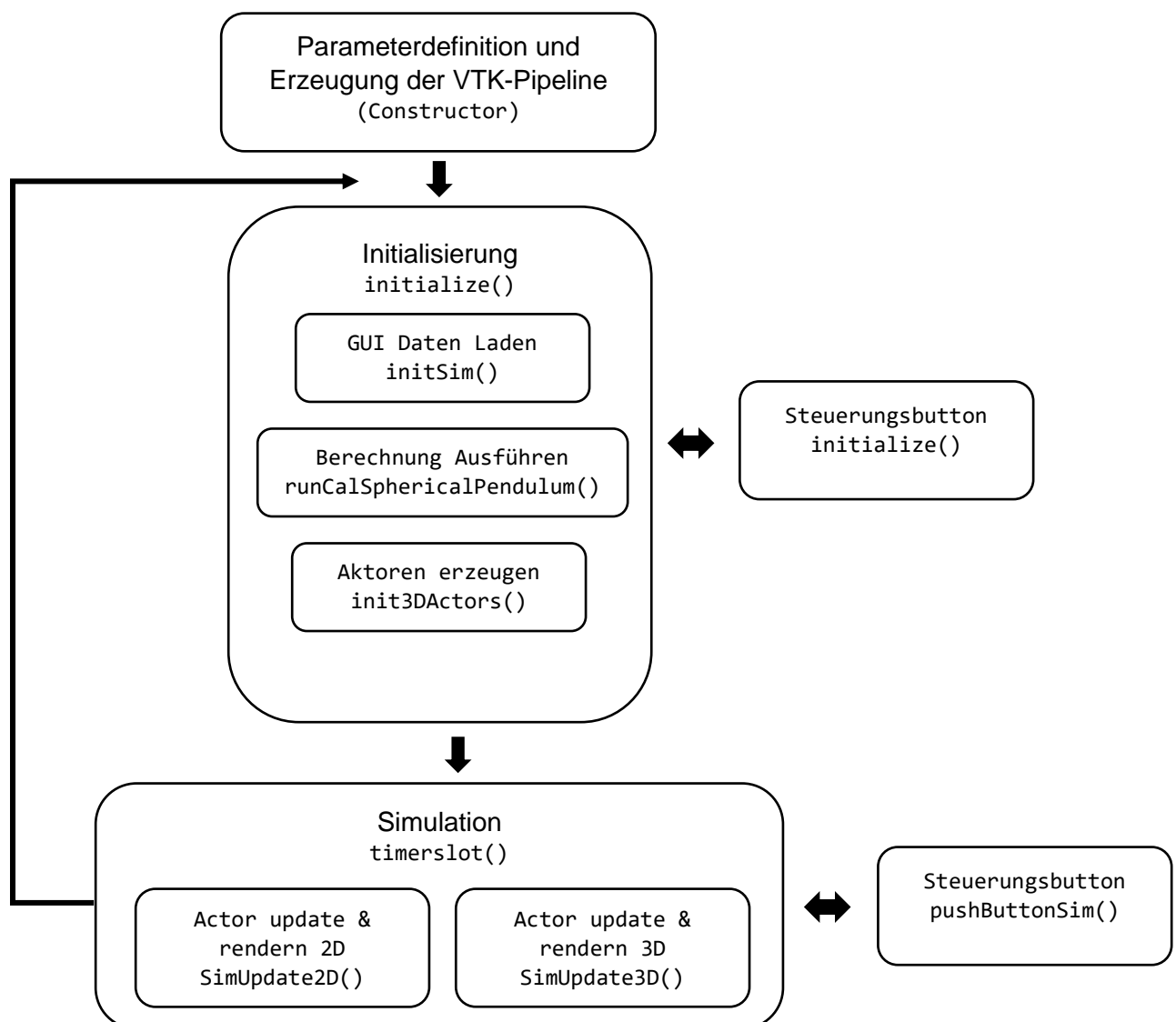


Abbildung 3-6 Übersicht des Programmablaufs

Dieser Ablauf ist als Ganzes im Konstruktor implementiert und wird ausgeführt, sobald ein Objekt im Main Programm angelegt wird.

Parameterdefinition und VTK-Pipeline

Die Klasse ist so aufgebaut, dass beim Ausführen des Konstruktors die nötigen Konstanten auf vorgegebene Werte definiert werden, welche für die weitere Verwendung im Programm notwendig sind. Sind diese festgelegt, so folgt im nächsten Schritt der Aufbau der VTK-Pipeline für die Visualisierungen im 2D und 3D. Das beinhaltet im Groben die Geometrie Sourcen, Mapper, Actor sowie die Definition ihrer Eigenschaften. Dazu werden bereits Funktionen wie `init3DActors()` und `changeColorDefault()` aufgerufen. Die Benutzeroberfläche ist damit startbereit.

Initialisierung

Im Teil der Initialisierung erfolgt nach Eingabe der Daten durch den Benutzer die noch fehlende Initialisierung des Programms. Dazu gehört das Zurücksetzen der

```
// Initialize Simulation and Render Window
void PendulumPainter::initialize() {
    /**-----
     * Description:  Global initialize method. All sub-initializations are called within this
     *               function.
     * Overview:    1. initSim()
     *               2. init3DActors()
     *               3. runCalSphericalPendulum();
     *
     * Button:      Initialize
     * -----*/

    // Call global init method
    initSim();

    // Start Calculation with sphericalPendulum class
    runCalSphericalPendulum();

    //Initialize Actors
    points3D->Initialize();
    points2D->Initialize();
    ren->RemoveActor(line3DActor);
    ren2D->RemoveActor(line2DActor);

    //Initialize ProgressBar
    this->ui->progressBar->reset();
    this->ui->progressBar->setMinimum(1);
    this->ui->progressBar->setMaximum(matCalData.size());
    this->ui->progressBar->setValue(0);

    // Create new Actors in 3D view according to current Ui Data
    init3DActors();

    //assembly->RotateZ(10);
    std::cout << "orientation:" << *assembly->GetOrientation() << endl;

    //resize camera view:
    ren->ResetCamera();

    //Update Qt Window
    this->ui->qvtkWidget3D->renderWindow()->Render();
    this->ui->qvtkWidget2D->renderWindow()->Render();

    QCoreApplication::processEvents();

    std::cout << "GIU data initialized!\n";
}
```

Abbildung 3-7 Quellcode zur Initialisierung der Simulationsschleife

Simulationsschleife, die Erzeugung der Berechnungsdateien mittels Berechnungs-Klasse sowie die Initialisierung und Anpassung der Aktoren. Abbildung 3-7 zeigt den dazugehörigen Quellcode.

Simulation

In der Simulationsschleife kommt ein von Qt zur Verfügung gestelltes Timer Konzept zum Einsatz. Das ist notwendig, um auch während der Simulation Interaktionen mit der Benutzeroberfläche zu ermöglichen. Es wird ein Intervall festgelegt, welches definiert wie häufig eine Funktion aufgerufen werden soll. In dieser Anwendung wird die Funktion `timerslot()` aufgerufen, welche ein Position Update der Aktoren der 2D und 3D Darstellung sowie des Linienzuges des gezeichneten Musters updatet.

Würde man hierfür eine konventionelle Schleife wie beispielsweise eine While-Schleife verwenden, so ist keine Interaktion während der Simulation mit der Benutzeroberfläche möglich. Grund dafür ist, dass der Prozess zuerst abgearbeitet werden muss und danach erst wieder die Abarbeitung des nächsten Prozesses möglich ist. Bei der Verwendung des Timers kann hingegen zwischen zwei Simulation Updates ein anderer Prozess wie beispielsweise die Verarbeitung eines Slots beim Drücken des Pause Buttons, welcher den Timer deaktiviert und somit die Simulation pausiert.

Nach Ausführen der Simulation kann zu jedem Zeitpunkt die Klasse wieder mit den aktuell gültigen Eingabeparameter initialisiert werden.

Abbildung 3-8 zeigt den Quellcode zum Starten und Beenden des Timers wenn der Button *Start/Stop* gedrückt wird.

```
// SIMULATION Start-Stop (currently only for Pendulum Motion)
void PendulumPainter::pushButtonSim() {
    /**-----
     * Description: Activates and deactivates the TIMER - timerslot() which continually calls the
     *               simulation increment update
     *
     * Button:       Start/Stop
     * -----*/

    if (run == false) {
        this->ui->pushButtonSim->setText("Stop");
        run = true;
        this->timer->start();
        std::cout << "Timer active, running " << endl;
    }
    else {
        run = false;
        this->ui->pushButtonSim->setText("Start");
        std::cout << "Timer active, NOT running " << endl;
    }
}
```

Abbildung 3-8 Quellcode zum Starten und Beenden des Timers

Abbildung 3-9 zeigt den Quellcode der Funktion `timerslot()`. Diese wird nur dann aufgerufen, wenn der Timer aktiv ist. Das ist dann der Fall, wenn der Start Button gedrückt wurde.

```
// Simulation LOOP called by QTimer
void PendulumPainter::timerSlot() {
    /**
     * Description: Timer call continually the simulaiton increment update to render the 2D and 3D
     * window. Timer is active als long as there is no interruption by pushButtonSim()
     * or the end of the simulaiton data.
     */

    while (run) {
        if (numIncr < matCalData.size()-1)
        {
            SimUpdate3D();
            SimUpdate2D();
            this->ui->qvtkWidget2D->renderWindow()->Render();
            this->ui->qvtkWidget3D->renderWindow()->Render();

            this->ui->progressBar->setValue(numIncr+2);
            Sleep(simSpeedms);
            QCoreApplication::processEvents();
            numIncr++;

            /*std::cout << "Step: " << numIncr
             << "\tAngleIncr.: [" << matCalData[numIncr][2] << ", "
             << matCalData[numIncr][3] << "]" "
             << "\tCoord.: [X: " << matCalData[numIncr][0] << " | Y: "
             << matCalData[numIncr][1] << "]" << endl;*/

        }
        else
        {
            std::cout << "Simulation finished with incement: " << numIncr << endl;
            this->timer->stop();
            run = false;
        }
    }
}
```

Abbildung 3-9 Quellcode der Funktionen welche bei aktiven Timer aufgerufen werden

Zusätzliche Funktionen

Zu den erwähnten Hauptfunktionen der Klasse PendulumPainter sind noch weite vom Umfang etwas kleinere Funktionen implementiert. Dazu gehören beispielsweise Funktionen zum Wechseln der Hintergrundfarbe der 2D und 3D Ansicht, des Zeichenpapiers und der Zeichenfarbe. Zudem gibt es noch eine Funktion zur Wiederherstellung der Standardeinstellung.

Damit das di erstellte 2D Zeichnung abgespeichert werden kann, wird die Funktion `saveImage()` aufgerufen. Diese ist einerseits über die Toolbar aufrufbar, kann aber alternativ auch über das Menü aufgerufen werden. Die vollständige Liste sowie die genauen Funktionsbezeichnungen sind aus Tabelle 3-1 zu entnehmen.

Nachfolgende Abbildung 3-10 zeigt als Ausschnitt aller Zusatzfunktionen die Funktion zur Speicherung der 2D Zeichnung im .png Format.

```

void PendulumPainter::saveImage() {
    /**
     * Description:  Function to save the drawn 2D image.
     *
     * Button:      Print-Button
     */

    // Select directory where to save .png
    QString imagePath = QFileDialog::getExistingDirectory(this, tr("Open Directory"), "/home",
                                                         QFileDialog::ShowDirsOnly | QFileDialog::DontResolveSymlinks);
    imagePath.append("/PendulumPainter.png");

    // Screenshot
    vtkNew<vtkWindowToImageFilter> windowToImageFilter;
    windowToImageFilter->SetInput(this->ui->qvtkWidget2D->renderWindow());
    windowToImageFilter->ReadFrontBufferOff();          // read from the back buffer
    windowToImageFilter->Update();

    //writing
    vtkNew<vtkPNGWriter> writer;;
    writer->SetFileName(imagePath.toLocal8Bit().data());
    writer->SetInputConnection(windowToImageFilter->GetOutputPort());
    writer->Write();

    // Textbox
    QMessageBox msgBox;
    msgBox.setText("Image Saved!");
    msgBox.exec();

    std::cout << "Image Saved to : " << imagePath.toStdString() << endl;
}

```

Abbildung 3-10 Quellcode zur Speicherung der 2D Zeichnung im .png Format

4 GUI

Die grafische Benutzeroberfläche wurde mittels Qt erstellt. Dabei erleichterte die Verwendung des Qt Designers, welcher auch von Visual Studio aus ausgeführt werden kann, die Erstellung deutlich. So können mit Drag and Drop, Elemente auf die Oberfläche gezogen werden, welche ein Objekt abbilden. Diese Objekte können dann im .cxx File der Klasse aufgerufen und individualisiert werden.

4.1 Aufbau der Benutzeroberfläche

Die folgende Auflistung zeigt einen Auszug der für die grafische Benutzeroberfläche aus Abbildung 4-1 verwendeten Elemente.

- Buttons
- Line Edit Elemente
- Text Edit Elemente
- Slider
- Progress Bar
- Spacer – notwendig für die Skalierung der GUI
- Combobox
- QWidget – zur Darstellung der in VTK erzeugten Pipeline

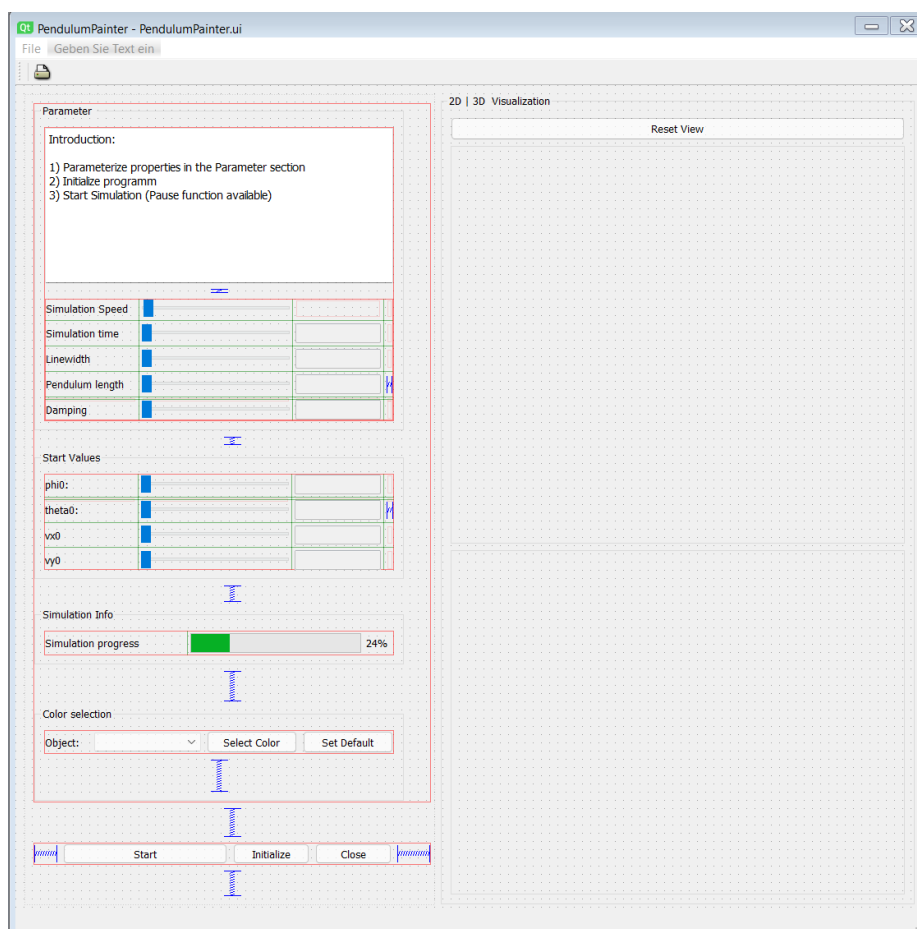


Abbildung 4-1 GUI - Qt Designer

Die in Abbildung 4-1 ersichtlichen roten Rahmen zeigen Layout Beziehungen und sind speziell für eine bestehende Anordnung beim Vergrößern oder Verkleinern der Anwendung notwendig.

In der folgenden Abbildung 4-2 ist das kompilierte Grafik-User-Interface (GUI) ersichtlich. Alle Eingabe Interaktionen beziehungsweise die Parameterdefinition des Anwenders finden im linken Bereich der GUI statt. Hier erfolgt auch die anschließende Initialisierung.

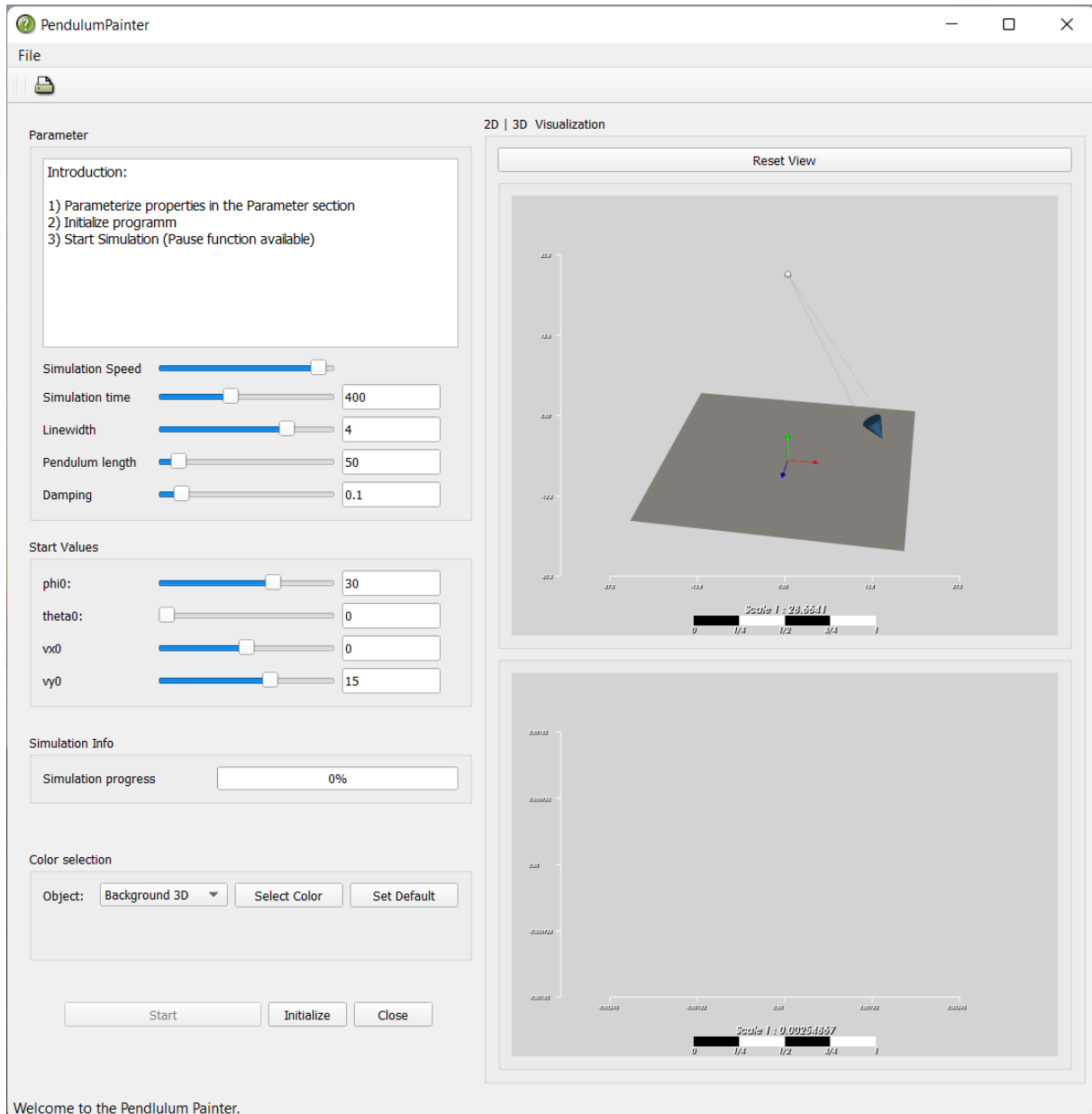


Abbildung 4-2 GUI kompiliert

4.2 Benutzereingabe und Interaktion

Im Folgenden werden im Einzelnen alle Funktionen der GUI beschrieben.

4.2.1 Grundparameter und Startwerte

Der Eingabebereich (Abbildung 4-3, links) erlaubt folgende Grundeinstellungen (Parameter):

- Simulation Speed – Simulationsgeschwindigkeit der Pendelbewegung
- Simulation time – Begrenzt die Berechnungszeit für das Differenzialgleichungssystem
- Linewidth – Dicke der gezeichneten Linie
- Pendulum length – Länge des Pendels
- Damping – Dämpfungswert des Pendels

Weiters müssen Startbedingungen (Start values) für die Pendelbewegung definiert werden. Dies wird durch die Pendelauslenkung und Startgeschwindigkeit, wie in ersichtlich, definiert:

- Φ_0 = Pendelauslenkung in Grad
- Θ_0 = Pendelauslenkungsposition in Grad
- v_{x0} = Startgeschwindigkeit in x-Richtung
- v_{y0} = Startgeschwindigkeit in y-Richtung

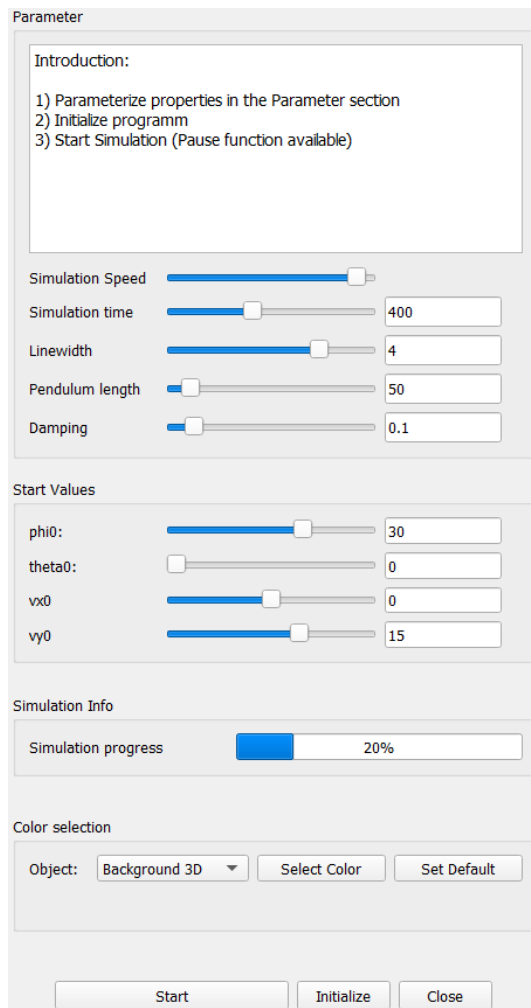


Abbildung 4-3 Eingabe

4.2.2 Farben ändern

Bei der Farbauswahl (Color Selection), in Abbildung 4-4 dargestellt, kann das Objekt ausgewählt werden, von welchem die Farbe geändert werden soll. Zur Auswahl stehen:

- Background 3D – Änderung der Hintergrundfarbe im 3D-Fenster
- Background 2D - Änderung der Hintergrundfarbe im 2D-Fenster
- Drawing paper- Änderung der Farbe des Zeichenplattes
- Painting color – Änderung der Farbe der gezeichneten Linie

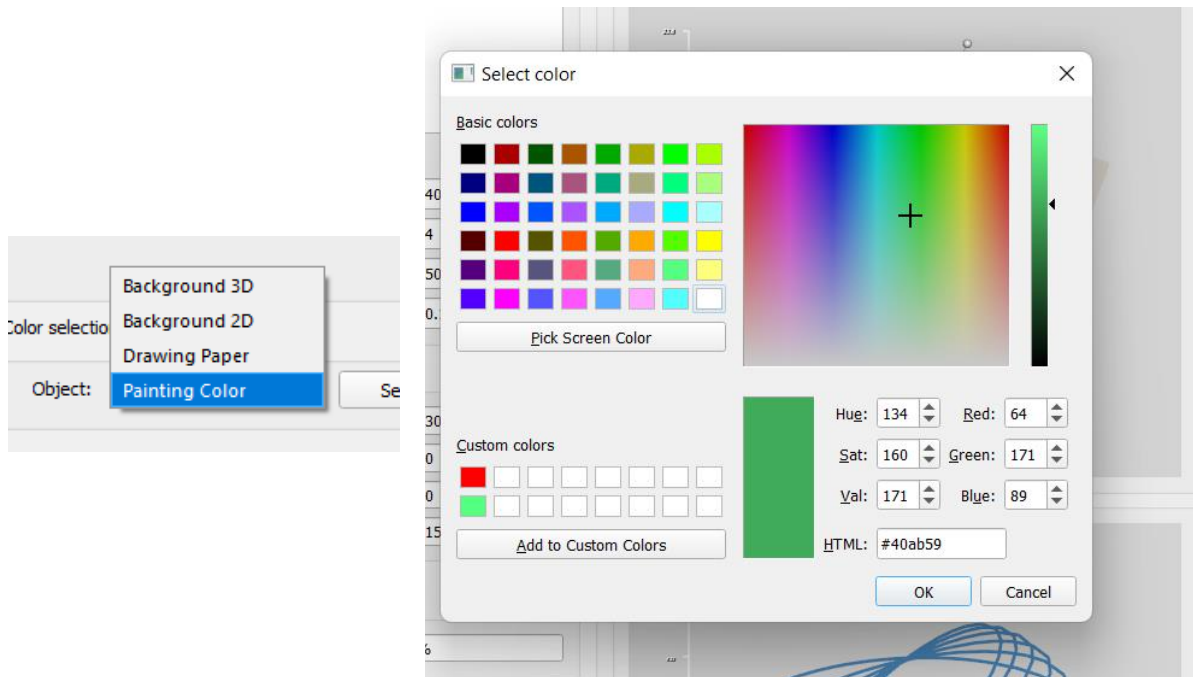


Abbildung 4-4 Übersicht Farbauswahl

4.2.3 Bild abspeichern

Über die Menübar beziehungsweise Toolbar ist ein Speicher-Symbol erreichbar. Führt man diese Funktion aus wird ein Bild von der gezeichneten Bildebene (2D-Ansicht) abgespeichert.

4.3 Animation

Die Animation der Pendelbewegung inklusive gezeichnetes Bild wird in einem 3D Fenster (Abbildung 4-5) dargestellt. In diesem Fenster ist interaktives drehen, verschieben oder zoomen möglich.

Darunter befindet sich ein weiteres Anzeigefenster, ein 2D Fenster. Dieses stellt nur die gezeichnete Bildtrajektorie in der Draufsicht dar. Hier ist interaktives zoomen bzw. verschieben möglich.

Die Ansichten können über einen Push-Button („Reset view“) wieder in ihre initiale Darstellung zurückgesetzt werden.

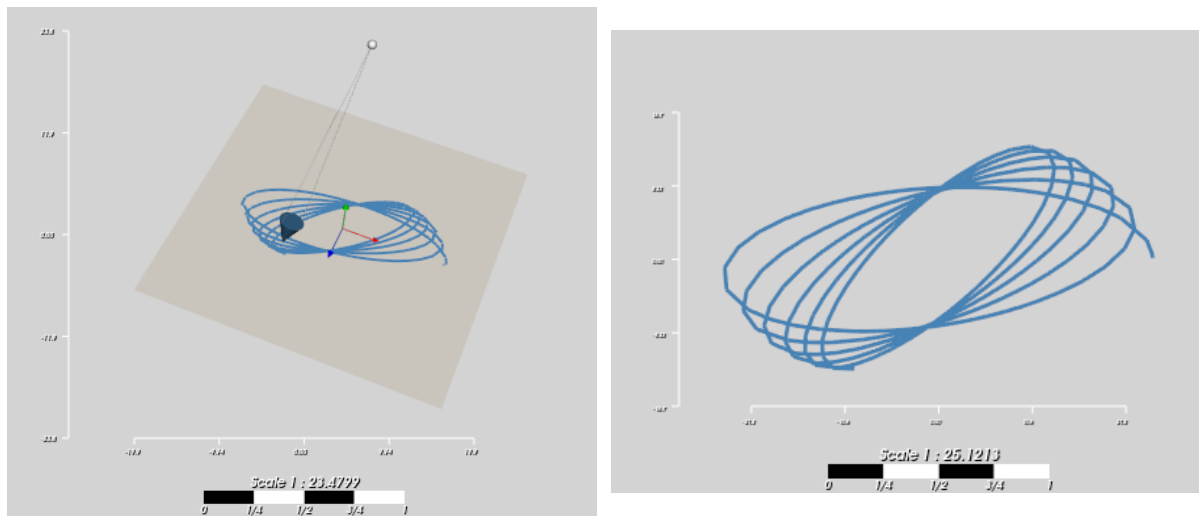


Abbildung 4-5 Links: Ansicht 3D, rechts: Ansicht 2D

5 Zusammenfassung

Zusammenfassend kann gesagt werden, dass VTK zusammen mit Qt eine sehr gute Möglichkeit bietet, um umfangreiche Visualisierungen mit überschaubarem Zeitaufwand erzeugen zu können.

Die Implementierung im Main erfolgt sehr einfach mittels weniger Zeilen. In Abbildung 5-1 ersichtlich der notwendige Qt und VTK-Code sowie die Anlage der Instanz PendulumPainter.

```
// Qt includes
#include <QtGui/QSurfaceFormat>
#include <QtWidgets/QApplication>
#include "QVTKOpenGLStereoWidget.h"

// Other includes
#include "PendulumPainter.h"

// Basics
using namespace std;
typedef vector<vector<double>> matrix;

extern int qInitResources_icons();

// Main
int main(int argc, char** argv)
{
    // needed to ensure appropriate OpenGL context is created for VTK rendering.
    QSurfaceFormat::setDefaultFormat(QVTKOpenGLStereoWidget::defaultFormat());

    // QT Stuff
    QApplication app(argc, argv);
    QApplication::setStyle("fusion");
    qInitResources_icons();

    // PendulumPainter Obeject
    PendulumPainter myPendulumPainter;
    myPendulumPainter.show();

    return app.exec();
}
```

Abbildung 5-1 Quellcode Main

Allerdings brachten einige Funktions-Implementierungen auch Probleme mit sich. Im Nachfolgenden wird kurz auf die Schlüsselstellen der Implementierung beider Klassen eingegangen.

Klasse Pendulum Painter

Eine der Schlüsselstellen in der Klasse Pendulum Painter war die Realisierung zum Pausieren der Visualisierung der Pendelbewegung. Diese konnte aber durch den in Kapitel 3.2.2 erläuterten Qt-Timer umgesetzt werden. Des Weiteren sind Probleme bei der inkrementellen Anpassung der Pendelposition (Simulationsupdate) aufgetaucht. Aufgrund der Tatsache, dass die Bewegungsgleichung des Pendels in einem von VTK abweichenden Koordinatensystem aufgestellt worden ist, konnten die Koordinaten nicht einfach übernommen werden. Das Problem konnte aber durch mehrmaliges Anpassen bzw.

Vertauschen der Koordinaten gelöst werden. Zuletzt traten noch Probleme bei der Interaktion in der 2D Darstellung auf. Durch die Verwendung der Klasse `vtkInteractorStyleImage` konnte der Interactor jedoch so angepasst werden, dass nur mehr zoomen möglich war und die Darstellung somit einen 2D Charakter hat.

Klasse Spherical Pendulum

Für die Lösung des ODE-Systems ist ein Lösungsalgorithmus notwendig. Da eine Selbstprogrammierung aufwendig wäre und nicht Ziel des Projektes war, wurde auf eine bestehende und in dem Anwendungsgebiet etablierte Bibliothek zurückgegriffen. Die Einarbeitung in diese Bibliothek (`odeint`) war doch aufwendig, da die Denkweise beziehungsweise Logik verschiedener Bibliotheken und deren Funktionen doch unterschiedlich ist. Weiters verkompliziert wurde dies noch durch die Vielzahl an implementierten ODE-Lösern. Mittels Internetrecherche, Durchsicht der Beschreibung sowie durcharbeiten der Beispielpcodes ist es gelungen die hier zugrunde liegende mathematische Aufgabenstellung entsprechend zu lösen und als Klasse aufzubauen.

Abschließend zeigt die Gegenüberstellung in der nachfolgenden Abbildung 5-2 links das initiale Bild von der einführenden Motivation. Mit den entsprechenden Startwerten simuliert dieses Programm die reale Bildentstehung nach, was rechts in der Abbildung zu sehen ist. Somit kann jede Startwertkombination simuliert werden und zeigt dem Anwender das zu erwartende Kunstwerk.

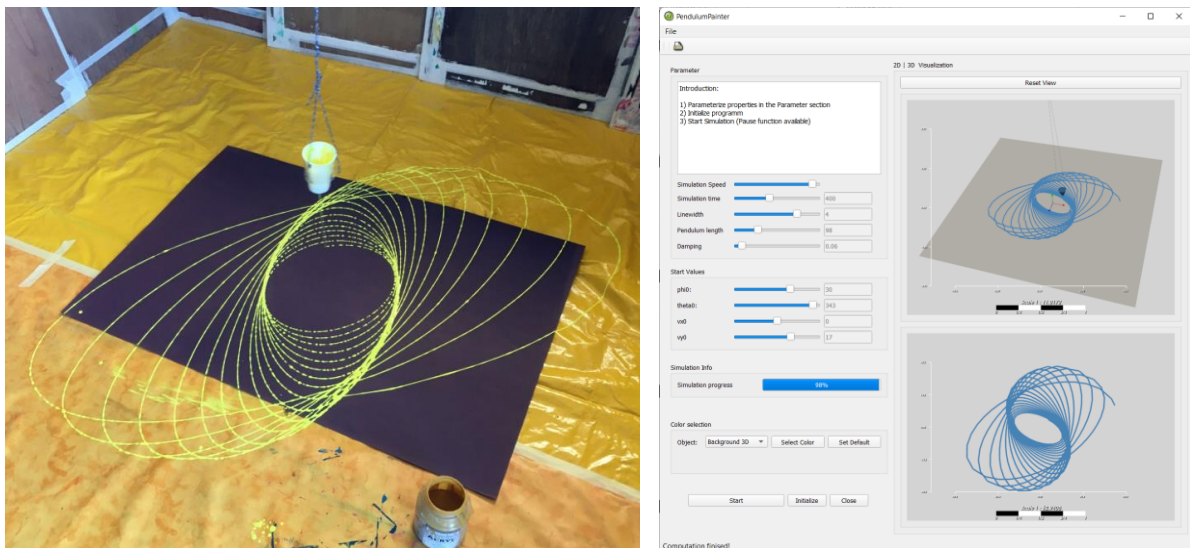


Abbildung 5-2 Vergleich Idee und Anwendung

6 Abbildungsverzeichnis

Abbildung 1-1 „Pendulum Painting“	3
Abbildung 1-2 Entwurf der grafischen Oberfläche und den darin enthaltenen Funktionen.	4
Abbildung 3-1 Sphärisches Pendel (auch Raum- oder Kugelpendel) (Steiner, 2015, p. 109)	7
Abbildung 3-2 Quellcode der Definition des ODE-Systems	8
Abbildung 3-3 Quellcode vom Löser des ODE-Systems	9
Abbildung 3-4 Quellcode des ODE-Definition	9
Abbildung 3-5 Signal-Slot-Konzept	10
Abbildung 3-6 Übersicht des Programmablaufs	11
Abbildung 3-7 Quellcode zur Initialisierung der Simulationsschleife	12
Abbildung 3-8 Quellcode zum Starten und Beenden des Timers	13
Abbildung 3-9 Quellcode der Funktionen welche bei aktiven Timer aufgerufen werden	14
Abbildung 3-10 Quellcode zur Speicherung der 2D Zeichnung im .png Format	15
Abbildung 4-1 GUI - Qt Designer	16
Abbildung 4-2 GUI kompiliert	17
Abbildung 4-3 Eingabe	18
Abbildung 4-4 Übersicht Farbauswahl	19
Abbildung 4-5 Links: Ansicht 3D, rechts: Ansicht 2D	20
Abbildung 5-1 Quellcode Main	21
Abbildung 5-2 Vergleich Idee und Anwendung	22

7 Tabellenverzeichnis

Tabelle 3-1 Beschreibung aller in der Klasse PendulumPainter vorhandenen Funktionen.....11