Stony Brook University

Design and Development of a NURBS and Dual Quaternion based extensible Motion Design Library and Software

A Thesis Presented
by

Prasad Dixit

to

The Graduate School
in partial fulfillment of the
Requirements
for the degree of

Master of Science
in
Mechanical Engineering

Stony Brook University
May 2009

Stony Brook University
The Graduate School
Prasad Dixit
We, the thesis committee for the above candidate for the
Master of Science degree,
hereby recommend acceptance of this thesis.

Dr. Anurag Purwar, Advisor,
Research Assistant Professor, Mechanical Engineering Department

Dr. Q. Jeffrey Ge,Co-Advisor
Professor, Mechanical Engineering Department

Dr. Yu Zhou, Chairman of Thesis Committee,
Assistant Professor, Mechanical Engineering Department

This thesis is accepted by the Graduate School.

Lawrence Martin
Dean of the Graduate School

Abstract of the Thesis

Design and Development of a NURBS and Dual Quaternion based extensible Motion Design Library and Software

by

Prasad Dixit

Master of Science

in

Mechanical Engineering

Stony Brook University

2009

Motion design via approximation and interpolation is of crucial importance in robotics, CNC tool path planning, computer graphics, computer aided geometric design, as well as in task specification in mechanism synthesis. Rational Motions, which produce rational as opposed to the transcendental trajectories are an attractive proposition since they integrate well with the existing NURBS based industry standard and are readily amenable to the applications of existing CAGD algorithms. By combining kinematics of rigid body motions with Non-Uniform Rational B-Spline (NURBS) geometry of curves and surfaces, methods are developed for computer aided design of rational motions.

This thesis deals with the development of an extensible object oriented software library and an application with a user friendly Graphical User Interface (GUI) for Non-Uniform Rational B-Splines (NURBS) and dual quaternion based computer aided design of rational motions. The motivation behind de-

veloping motion design library (abbreviated as MDL) is to provide a common platform with a comprehensive set of tools for implementing myriads of motion design algorithms to researchers, students, professors and other professionals working in the related areas. Such a platform would allow them to focus on the implementation of their own algorithms without worrying and spending time on developing display, printing, and visualization routines. A comprehensive set of functions which are dedicated to carry out various operations on dual quaternions, simple quaternions, dual numbers, homogeneous and regular matrices are developed which will assist developers to focus more on their own motion design algorithms, rather than on development of such routines.

An application called MoDes is also developed based on this library that can be used to plot motion, navigate in 3D space to examine the plotted motion, interactively modify and fine tune the motion by setting various parameters, and to generate good quality images of plotted motion for publication into research articles and reports.

There are few free and commercial softwares available which deal with motion design and mechanism synthesis problems in general. This work hopes to compliment the existing software systems by providing a library and and an application for design, manipulation and visualization of rigid body motions.

Table of Contents

List of Figures

List of Tables

ACKNOWLEDGEMENTS

I would like to express my sincere respect and gratitude to my research advisor, Professor Anurag Purwar, for his guidance and support through out my academic study and research. During the research work for this thesis , Dr. Purwar shared his knowledge in many areas, provided inspiration and was ready to help whenever I needed his advice. I am fortunate to have such a kind, knowledgeable and passionate advisor in my academic life. This thesis would have never reached fruition had it not been for the excellent ideas, intuition, and the thoughts that Professor Anurag Purwar shared with me during the course of this research. I am much grateful to him for this.

I would like to thank Prof. Yu Zhou for agreeing to chair my thesis committee, and Prof. Jeff Ge for helping me as my co-advisor and for being on my committee despite their busy schedule. I also appreciate the valuable comments of my committee members during the completion of this thesis. In general, I am thankful to all of my teachers from Mechanical Engineering, and Computer Science department at Stony Brook university.

I am very thankful Maryann, our ex-graduate program secretary and Diane, the graduate program secretary, who made sure that I was getting paid on time and always welcomed my visits to her office with a smile. Thanks are also due to Melissa, the Asst. to Chair, Augusta, the department secretary, and Erin Keffeler, the adviser to international faculty and scholars who took care of administrative hurdles that I faced during my instructional days and was always willing to help me.

I would also like to thank my parents and my brother for always having

faith in me and supporting me when I need it. I must also say that the company of my close friends I got during my stay at Stony Brook University made it all worth.

Overall, if my life as a graduate student in the last two and half years has been educating, inspiring, and pleasant, then it is thanks to Anurag Purwar, my advisor and mentor.

Chapter 1

Introduction

In recent years, there have been considerable efforts in bringing together kinematics and Computer Aided Geometric Design (CAGD) to develop methods for motion design in CAD environment. This thesis deals with the development of an extensible object oriented software library and an application with a user friendly Graphical User Interface (GUI) for Non-Uniform Rational B-Splines (NURBS) and dual quaternion based computer aided design of rational motions. In this introductory chapter, motivation behind developing motion design library is presented followed by a discussion on main contributions of this thesis and background material. Implemented motion design algorithms, object oriented framework adopted for this library, and existing softwares in the area of motion design and mechanism synthesis are also discussed in brief.

The motivation behind developing Motion Design Library (abbreviated as MDL) is to provide a common platform with a comprehensive set of tools for implementing myriads of motion design algorithms to researchers, students, professors and other professionals working in the related areas. Such a plat-

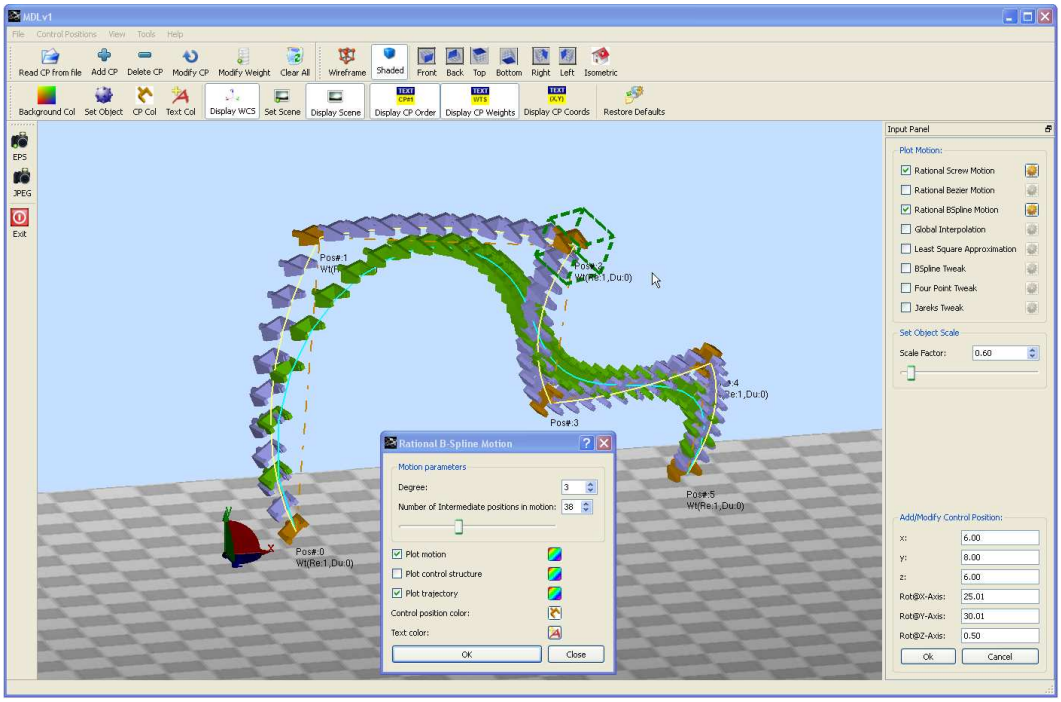


Figure 1.1: A screenshot of the MoDes application.

form would allow them to focus on the implementation of their own algorithms without worrying and spending time on developing display, printing, and visualization routines. The developer just needs to explore this neatly documented library and choose the appropriate functions to carry out desired operation. An application called MoDes is also developed based on this library that can be used to plot motion, navigate in 3D space to examine the plotted motion, interactively modify and fine tune the motion by setting various parameters, and to generate good quality images of plotted motion for publication into research articles and reports. Figure 1.1 shows a screenshot of the MoDes.

Kinematics (Reuleaux [1], Bottema and Roth [2], McCarthy [3], Hunt [4],

Angeles [5]) deals with the theory of movement of rigid bodies irrespective of the cause of the motion, while the field of Computer Aided Geometric Design (CAGD) (Farin [6, 7], Farin et al. [8], Hoschek and Lasser [9], Piegl and Tiller [10], Gallier [11]) concerns itself with the geometry of curves and surfaces for CAD. The NURBS (Non-Uniform Rational B-Spline) geometry of curves and surfaces in CAGD has emerged as the de facto industry standard because it can handle both standard analytical shapes (conics, quadrics, surfaces of revolution, etc.), and free-form shapes and offers computationally fast and stable algorithms for shape design and modification (see Farin [12], Piegl [13, 14, 15, 16, 17, 18, 19, 20, 21], Piegl and Tiller [22, 23], Tiller [24], Böhm [25], Boehm [26]). Lately, researchers have sought to combine kinematics of rigid body motions with Non-Uniform Rational B-Spline (NURBS) geometry of curves and surfaces to develop methods for computer aided design of motions. These CAD methods for motion design find applications in animation in computer graphics (key frame interpolation), trajectory planning in robotics (taught-position interpolation), spatial navigation in virtual reality, computer aided geometric design of motion via interactive interpolation, CNC tool path planning, and task specification in mechanism synthesis.

The influential work of Shoemake [27], introduced quaternions to computer graphics and showed how quaternions can be combined with the De Casteljau algorithm for animating rotations. Since then, there have been considerable efforts to combine motion representation theory with curve design techniques in Computer Aided Geometric Design (CAGD) for Cartesian motion synthe-

sis. On the motion representation side, a common approach is to decompose a spatial displacement into a translation of a point O and a rotation about O. In this way, rotation and translation components can be interpolated separately. Since the interpolation of translations is straightforward, the focus has been on the interpolation of rotations for which many quaternion-based schemes have been developed (Shoemake [27], Pletinckx [28], Kim and Nam [29]). Another approach is to consider a spatial displacement as one complete entity and as equivalent to a screw displacement. This has led to the development of dual-quaternion as well as Lie-group based methods for constructing spatial motions (see Ge and Ravani [30, 31, 32], Park and Ravani [33], Zefran and Kumar [34]). This approach results in spatial motions that are invariant with respect to change of fixed and moving reference frames. For adapting curve design techniques in CAGD for motion synthesis, two fundamentally different approaches have emerged. The first approach, originated by Shoemake [27] and followed by Pletinckx [28], Barr et al. [35], Wang and Joe [36], Nielson and Heiland [37], Kim and Nam [29], considers a unit quaternion as defining a unit hypersphere and studies the problem of rotation interpolation as a curve design problem on the surface of the hypersphere. This hyperspherical approach has been extended by Ge and Ravani ([38], [30]) as well as Ge and Kang [39] for complete motion generation (including both rotations and translations) by synthesizing curves on a unit dual hypersphere. Dual projective three space is termed as the Image Space of spatial kinematics by Ravani and Roth [40]. In the second approach, a spatial displacement is represented by an

image point in image space; a one-parameter motion is represented by an one-parameter curve (image curve) in image space. If the image curve is defined by a global or a piecewise polynomial designed using Bézier or B-spline interpolating functions, the corresponding motion in three-space is a rational Bézier or B-spline motion. This is the approach pioneered by Ge and Ravani [30] and then followed by Jüttler [41, 42], Jüttler and Wagner [43], and Wagner [44, 45], Purwar and Ge [46]. The rational motions generated using this approach by applying standard CAGD techniques are easy to compute and efficient, possess a control structure similar to the control polygons of Bézier or B-spline curves or surfaces, have subdivision property, and are coordinate frame invariant. Moreover, the approach followed by Ge and Ravani [30] and Purwar and Ge [46] gives an intrinsic control structure that allows for interactive motion design (Jüttler and Wagner [47]).

This work has focussed on developing a framework for an extensible software library as well as an application which can serve as a common platform for various NURBS and dual quaternion based motion design algorithms. However, the library is not strictly restricted to NURBS class of motion and other types of motion such as those based on simple subdivision principle are also integrated into the library.

In my work, I have implemented eight motion design algorithms, under various classes of motions. Rational Screw Motion, Bézier motion, B-spline motion come under the fundamental motions. My implementation of above mentioned algorithms also explored the effects of dual weights and reparametriza-

tion for path invariance. Motion fitting is a critical problem in the field of motion design. Designers often seek a motion interpolating through given key frames or approximating the given set control positions. Global Interpolation to Position Data and Least Square Motion Approximation are the two algorithms which touches the problem of motion fitting. With a view that this library can be used as educational tool by the professors and researchers working in the field of CAGD, I thought it will be good to add few motion design schemes which are simple to implement and understand. The motive behind this was to generate interest amongst the students about this subject in general. So with that point in mind, three simple subdivision based motion design schemes namely, B-spline tweak, 4-point tweak and Jarek's Tweak from a paper by Rossignac (Rossignac [48]) are added.

In development of this library, object oriented methodology has been adopted. The object oriented programming , often referred to as OOP helps to formulate the problem in a better way giving high reliability, adaptability and extensibility to the applications. A major advantage of OOP is code reusability. OOP provides a clear modular structure for programs which makes it good for defining abstract data-types where implementation details are hidden and the unit has a clearly defined interface. It makes it easy to maintain and modify existing code as new objects can be created with small differences to existing ones. It also provides a good framework for code libraries where supplied software components can be easily adapted and modified by the programmer. In an effort to develop this library, I have written an object oriented program which

extends more than 12,000 lines. The library is developed using following tools:

1. Microsoft Visual C++ : A comprehensive Integrated Development Environment (IDE) for development, debugging and deployment of software applications

2. Qt from Trolltech: A framework for developing applications and user interfaces (UI).

3. OpenGL: A premier environment for developing portable, interactive 2D and 3D graphics applications. OpenGL is the most widely used and supported 2D and 3D graphics application programming interface (API).

4. GLUT : An OpenGL Utility Toolkit, a window system independent toolkit for writing OpenGL programs.

This library can be broadly divided into three parts. QLib which stands for quaternion library, MLib which stands for motion library and GUI which is abbreviation for Graphical User Interface. QLib and MLib are strongly interconnected with each other. However, lot of efforts has been put to keep GUI separate from QLib and MLib. The reason behind such step was to give the developer freedom to develop a graphical user interface from free and common GUI toolkits such as GLUI, GLUT, FLTK etc., instead of Qt, if necessary.

There already exists a few free and commercial softwares which deal with motion design and mechanism synthesis problems in general. Computer-aided linkage design software computes the form of a device from a function specified by the designer. Function-to form linkage design tools have recently

been commercialized by SyMech Inc., (SyMech [49]) and Heron Technologies, (WATT [50]). These systems compute the dimensions for several planar linkage topologies, which are combinations of 4R closed chains and 2R open chains, given functional specifications provided by the user. The first of this type of software system was KynSyn (Kaufman [51]) which focussed solely on the 4R planar topology which was dimensioned to guide a body through a specified set of positions R denotes a revolute or hinged joint. RECSYN (Waldron and Song [52]) and LINCAGES (Erdman and Gustafson [53]) added features that simplified the design process but again focussed on planar 4R and later planar 6R linkages. The linkage design software Sphinx (Larocelle et al. [54]) and later SphinxPC (Ruth and McCarthy [55]), extended Kaufmans strategy to 4R linkages that move on a sphere providing the first design tool for the design of linkages for spherical movement. Furlong et al. [56] used virtual reality to assist the designer's specification of desired function of a spherical 4R linkage as well as to evaluate the resulting computed device. The first design software for a true spatial linkage was Larochelle's SPADES software (SPADES [57]) which computed a spatial 4C closed chain to guide a body through four spatial positions-C denotes a cylindric joint which allows rotation about and sliding along a given axis. Synthetica (Synthetica [58])is a Java-based software developed by Prof. McCarthy's group at University of California, Irvine. It is used for the synthesis, visualization, analysis and simulation of spatial linkages. This work hopes to compliment the existing software systems by providing a motion design library and and an application for design, manipulation and

visualization of rigid body motions.A comprehensive set of functions which are dedicated to carry out various operations on dual quaternions, simple quaternions, dual numbers, homogeneous and regular matrices are developed which will assist developers to focus more on their own motion design algorithms, rather than on development of such routines.

The rest of the thesis is organized as follows. Chapter 2 deals with geometric and kinematic fundamentals that are necessary for the development of this thesis. It also reviews and studies representations of spatial displacements in three dimensional space with an emphasis on dual quaternion representation. Chapter 3 discusses the motion design schemes implemented so far in this motion design library, MDL. In this chapter, basically algorithms for three major categories of motion design problems are discussed. First, the fundamental motion types which includes rational screw motion, rational Bézier motion, and rational B-spline motion are discussed. Second, motion fitting, under which global interpolation to position data and least square motion approximation are elaborated. Lastly, a subdivision based motion design scheme, which includes B-spline tweak, 4-point tweak and Jarek's tweak based on a paper by Rossignac [48] is discussed. Chapter 4 is a guide to using MoDes. It describes the set of convenient tools available specific to interactively designing, manipulating, and plotting motion. Chapter 5 serves as guideline to developers who wish to extend the motion design library. The final chapter summarizes the work of this research and makes a few salient points regarding the future development of this work. The appendix contains notes on installing MDL and

setting up Visual Studio's environment for compiling this library. A detailed documentation of all the classes, functions and variables in this library can also be located in appendix of this thesis.

Chapter 2

Geometric and Kinematic Fundamentals

This chapter deals with geometric and kinematic fundamentals that are necessary for the development of this thesis. Several representations of spatial displacements are discussed with emphasis on dual quaternion based representation. The reason behind emphasis on dual quaternion is that algorithms implemented in this library are based on dual quaternion representation of spatial displacements. Kinematic mapping of NURBS curve in dual quaternion space to NURBS motion in three-space is explained followed by elaboration of point trajectory and affine control structure of a one parameter rational Bézier motion.

2.1 Representation of Spatial Displacements

In studying spatial displacements of an object in Euclidean three-space (E^3), it is convenient to attach a Cartesian coordinate system (or frame) M to the moving object and another Cartesian coordinate frame F to the fixed space

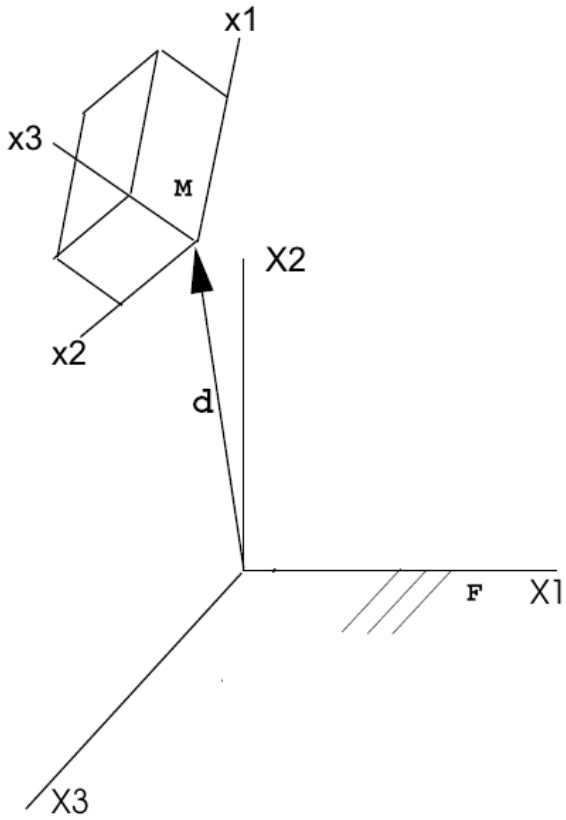


Figure 2.1: Spatial displacement of an object in three space E^3.

E^3 and study the position and orientation of the frame M with respect to F. The position of M with respect to F is given by the vector $\mathbf{d} = (d_1, d_2, d_3)$ from the origin of F to the origin of M. The orientation of M relative to F is given by a 3×3 rotation matrix $[R]$.

A spatial displacement is most commonly represented as a rigid transformation from M to F in terms of point coordinates:

$$\begin{bmatrix} \mathbf{X} \\ 1 \end{bmatrix} = \left[\begin{array}{ccc|c} & [R] & & \mathbf{d} \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} \tag{2.1}$$

where $\mathbf{X}(X1, X2, X3)$ and $\mathbf{x}(x1, x2, x3)$ are vectors whose scalar components are the Cartesian coordinates of the point as measured in F and M, respectively (Refer Fig. 2.1). The use of such matrix representation, however is not convenient when dealing with the problem of synthesizing a rational motion that interpolates or approximates a set of displacements. One of the main obstacles is to the issue of preserving the orthogonality of the rotation matrix in the interpolation/approximation process (Fillmore [59], Röschel [60]). It has been recognized that an effective way of dealing with the problem is to use quaternions (Shoemake [27]) and dual quaternions (Ge and Ravani [31]). Quaternion (Hamilton [61, 62], Bottema and Roth [2], Waerden [63], Arunachalam [64], Cheng and Gupta [65], Pervin and Webb [66]) representation of rotation as a quadruple of Euler parameters [1] is compact (requires only four numbers), computationally and storage-wise most efficient (Eberly [67]), unique up to a scale factor, and coordinate frame independent. It also carries a clear geometric meaning, produces singularity free transformations, and concatenates and interpolates nicely (Dam et al. [68], Vicci [69]).

In the following sections, we will review the concepts of quaternions and dual quaternions in so far necessary for development of the thesis.

2.1.1 Unit Quaternions as Rotation and Scalers as Weights

A unit quaternion is an elegant tool to represent a rotation. Any rotation $[R]$ in E^3 has a fixed axis as well as an angle of rotation. Let $\mathbf{s} = (s_1, s_2, s_3)$ denote

[1] Euler parameters are not to be confused with the Euler angles described earlier.

the unit vector along the axis of rotation and θ be the angle of rotation. They can be used to define the so-called Euler-Rodrigues parameters of a rotation, $\mathbf{q} = (q_1, q_2, q_3, q_4)$ where:

$$q_1 = s_1 \sin\frac{\theta}{2}, \quad q_2 = s_2 \sin\frac{\theta}{2}, \quad q_3 = s_3 \sin\frac{\theta}{2}, \quad q_4 = \cos\frac{\theta}{2} \tag{2.2}$$

The four Euler parameters satisfy the relation

$$q_1^2 + q_2^2 + q_3^2 + q_4^2 = 1$$

Therefore, $\mathbf{q}$ can be thought of as a point lying on unit 3-sphere (S^3) embeded in a four dimensional place.

The rotation matrix $[R]$ can be recovered from Euler parameters using (Bottema and Roth [2]):

$$[R] = \frac{1}{S^2}\begin{bmatrix} q_4^2+q_1^2-q_2^2-q_3^2 & 2(q_1q_2-q_4q_3) & 2(q_1q_3+q_4q_2) \\ 2(q_2q_1+q_4q_3) & q_4^2-q_1^2+q_2^2-q_3^2 & 2(q_2q_3-q_4q_1) \\ 2(q_3q_1-q_4q_2) & 2(q_3q_2+q_4q_1) & q_4^2-q_1^2-q_2^2+q_3^2 \end{bmatrix} \tag{2.3}$$

where $S^2 = q_1^2 + q_2^2 + q_3^2 + q_4^2$.

From the above, it is clear that the rotation matrix $[R]$ remains the same after multiplying each of the Euler-Rodrigues parameters by a scaler $w(w \neq 0)$. This means that $\mathbf{q}$ can be treated as homogeneous coordinates of rotation. In this thesis, we denote non-unit quaternion of homogeneous coordinates by $\mathbf{Q} = w\mathbf{q}$. It is clear that the homogeneous quaternion $\mathbf{Q}$ represents one and the same rotation irrespective of the choice of non zero weight w.

2.1.2 Dual Quaternion representation of Spatial Displacements and Dual Numbers as Dual Weights.

On similar lines to the homogeneous quaternion $\mathbf{Q}$, the translation vector $\mathbf{d}$ can also be homogenized and written in a quaternion form to obtain ($\mathbf{D} = (w_0\mathbf{d}, w_0)$). Jüttler [41], Jüttler and Wagner [43] used this set of eight homogeneous parameters (**Q,D**) for computer aided design of rational motions. While this formulation allows direct application of existing CAGD techniques to motion design, the resulting motions are not completely reference-frame invariant and depend on the choice of the origins of the reference frames. In this thesis we follow McCarthy [3] and Ge and Revani [31] and use, a slightly modified version of Study's Soma parameters (Bottema and Roth [2], Study [70]) to represent the spatial displacement. Study's parametersare given by another set of eight homogeneous parameters ($\mathbf{Q}, \mathbf{Q^0}$), where $\mathbf{Q} = (Q_1, Q_2, Q_3, Q_4)$ represents the quaternion of homogeneous Euler parameters of rotation and $\mathbf{Q}^0 = (Q_1^0, Q_2^0, Q_3^0, Q_4^0)$ is another quaternion whose components are given by

$$\begin{bmatrix} Q_1^0 \\ Q_2^0 \\ Q_3^0 \\ Q_4^0 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 0 & -d_3 & d_2 & d_1 \\ d_3 & 0 & -d_1 & d_2 \\ -d_2 & d_1 & 0 & d_3 \\ -d_1 & -d_2 & -d_3 & 0 \end{bmatrix} \begin{bmatrix} Q_1 \\ Q_2 \\ Q_3 \\ Q_4 \end{bmatrix} \tag{2.4}$$

The translation vector $\mathbf{d} = (d_1, d_2, d_3)$ can be recovered from (2.4) in terms of $(\mathbf{Q}, \mathbf{Q}^0)$ by using the following

$$\mathbf{d} = \frac{2}{S^2} \begin{bmatrix} Q_4^0 Q_1 - Q_1^0 Q_4 + Q_2^0 Q_3 - Q_3^0 Q_2 \\ Q_4^0 Q_2 - Q_2^0 Q_4 + Q_3^0 Q_1 - Q_1^0 Q_3 \\ Q_4^0 Q_3 - Q_3^0 Q_4 + Q_1^0 Q_2 - Q_2^0 Q_1 \end{bmatrix}, \tag{2.5}$$

where $S^2 = Q_1^2 + Q_2^2 + Q_3^2 + Q_4^2$.

It is instructive to note that $(\mathbf{Q}, \mathbf{Q^0})$ serve as homogeneous coordinates of spatial displacements since multiplying them by a non zero scaler yields the same rotation matrix and translation vector $\mathbf{d}$.

Study parameters can also be written in dual vector form as $(\hat{\mathbf{Q}} = \mathbf{Q} + \epsilon\mathbf{Q}^0)$, where ϵ denotes the dual unit (see Bottema and Roth [2] for details on dual number). In quaternion form, Eqs. (2.4) and (2.5) can be written more concisely as follows, respectively:

$$\mathbf{Q} = \frac{1}{2}\mathbf{dQ}. \tag{2.6}$$

$$\mathbf{d} = \frac{(\mathbf{Q^0})\mathbf{Q}^* - \mathbf{Q}(\mathbf{Q^0})^*}{\mathbf{QQ}^*}. \tag{2.7}$$

where $\mathbf{d}$ is a vector quaternion, which has no scaler part, and $\mathbf{Q}^* = (-Q_1^0, -Q_2^0, -Q_3^0, Q_4^0))$ is the conjugate of $\mathbf{Q}$ such that $(\mathbf{QQ}^* = Q_1^2 + Q_2^2 + Q_3^2 + Q_4^2)$. Note that Eq. (2.5) or Eq. (2.7) can be used to recover $\mathbf{d}$ from $\mathbf{Q}$ and $(\mathbf{Q^0})$ even when they do not satisfy the well known Plücker condition:

$$Q_1Q_1^0 + Q_2Q_2^0 + Q_3Q_3^0 + Q_4Q_4^0 = 0. \tag{2.8}$$

However, when the dual components satisfy the above Plücker condition, Eq. (2.7) reduces to following well known equation

$$\mathbf{d} = \frac{\mathbf{2}(\mathbf{Q^0})\mathbf{Q}^*}{\mathbf{QQ}^*}. \tag{2.9}$$

which follows directly from Eq. (2.6).

An alternative way of defining a dual quaternion is based on the concept of screw displacements. In this case, a dual vector is used to define the screw axis

and a dual angle defines the angle of rotation about the axis and a dual angle defines the angle of rotation about the axis and the distance of translation along the axis. Let the screw axis be represented by a unit dual vector $\hat{\mathbf{s}} = (\hat{s}_1, \hat{s}_2, \hat{s}_3)$, and the dual angle be denoted by $\hat{\theta} = \theta + \epsilon h$. Then the four dual components of a dual quaternion can be given by so-called dual Euler parameters (Bottema and Roth [2], McCarthy [3]):

$$\hat{q}_1 = \hat{s}_1 \sin\frac{\hat{\theta}}{2}, \quad \hat{q}_2 = \hat{s}_2 \sin\frac{\hat{\theta}}{2}, \quad \hat{q}_3 = \hat{s}_3 \sin\frac{\hat{\theta}}{2}, \quad \hat{q}_4 = \cos\frac{\hat{\theta}}{2}. \tag{2.10}$$

The four dual Euler parameters satisfy the relation

$$\hat{q}_1^2 + \hat{q}_2^2 + \hat{q}_3^2 + \hat{q}_4^2 = 1. \tag{2.11}$$

The resulting dual quaternion $\hat{\mathbf{q}} = (\hat{q}_1, \hat{q}_2, \hat{q}_3, \hat{q}_4)$, is therefore a unit dual quaternion. Ravani and Roth [40] showed that an important advantage of dual quaternion formulation of spatial movements is that it is invariant with resect to change of both the moving and the fixed reference frames.

The dual orthogonal matrix $[\hat{R}]$ (McCarthy [3])can be parameterized with dual Euler parameters by

$$[\hat{R}] = \frac{1}{\hat{S}^2}\begin{bmatrix} \hat{q}_4^2+\hat{q}_1^2-\hat{q}_2^2-\hat{q}_3^2 & 2(\hat{q}_1\hat{q}_2-\hat{q}_4\hat{q}_3) & 2(\hat{q}_1\hat{q}_3+\hat{q}_4\hat{q}_2) \\ 2(\hat{q}_2\hat{q}_1+\hat{q}_4\hat{q}_3) & \hat{q}_4^2-\hat{q}_1^2+\hat{q}_2^2-\hat{q}_3^2 & 2(\hat{q}_2\hat{q}_3-\hat{q}_4\hat{q}_1) \\ 2(\hat{q}_3\hat{q}_1-\hat{q}_4\hat{q}_2) & 2(\hat{q}_3\hat{q}_2+\hat{q}_4\hat{q}_1) & \hat{q}_4^2-\hat{q}_1^2-\hat{q}_2^2+\hat{q}_3^2 \end{bmatrix}, \tag{2.12}$$

where $\hat{S}^2 = \hat{q}_1^2 + \hat{q}_2^2 + \hat{q}_3^2 + \hat{q}_4^2$.

The unit dual 3-sphere is the spherical model of the projective dual 3-space. Let $\hat{\mathbf{Q}} = (\hat{Q}_1, \hat{Q}_2, \hat{Q}_3, \hat{Q}_4)$ denote a general dual quaternion. Let $\hat{\mathbf{q}} = (\hat{q}_1, \hat{q}_2, \hat{q}_3, \hat{q}_4)$ denote a unit dual quaternion and let $\hat{w} = w + \epsilon w^0$ be the non

pure dual number. Then we have $\hat{\mathbf{Q}} = \hat{w}\hat{\mathbf{q}}$. In view of (2.12), it is clear that two dual quaternions $\hat{\mathbf{Q}}$ and $\hat{\mathbf{q}}$ represent one and the same orthogonal matrix and therefore, they represent same spatial displacement. Ravani and Roth [40] considered $\hat{\mathbf{Q}} = (\hat{Q}_1, \hat{Q}_2, \hat{Q}_3, \hat{Q}_4)$ as a set of four homogeneous dual coordinates that define a point in a projective dual 3-space $\hat{P}^3$, called the image space of spatial displacements. Thus, we may refer to $\hat{\mathbf{Q}}$ as a homogeneous dual quaternion.

2.1.3 Kinematic mapping

This thesis uses dual quaternion representation for spatial displacements. In geometry, "complex objects" such as spherical or spatial displacements in three-dimensional Euclidean space or lines in three-dimensional projective spaces are treated as points of a higher dimensional space via a special mapping(Klein and Blaschke [71], Stachel [72], Rath [73, 74]). Using this interpretation, lines in a real projective three-space map as elements of unit hypersphere or real projective three space, and following dual quaternion representation, spatial displacements map as elements of dual projective three space or unit dual hypersphere. Dual projective three space is termed as the Image Space of spatial kinematics by Ravani and Roth [40]. In this way, a spatial displacement is represented by an image point in image space; a one-parameter motion is represented by an one-parameter curve (image curve) in image space. If the image curve is defined by a global or a piecewise polynomial designed using Bézier or B-spline interpolating functions, the corresponding motion in

three-space is a rational Bézier or B-spline motion. This is the approach pioneered by Ge and Ravani [30] and then followed by Jüttler [41, 42], Jüttler and Wagner [43], and Wagner [44, 45], Purwar and Ge [46]. The rational motions generated using this approach by applying standard CAGD techniques are easy to compute and efficient, possess a control structure similar to the control polygons of Bézier or B-spline curves or surfaces, have subdivision property, and are coordinate frame invariant. Moreover, the approach followed by Ge and Ravani [30] and Purwar and Ge [46] gives an intrinsic control structure that allows for interactive motion design (Jüttler and Wagner [47]).

2.1.4 Point trajectory and affine control structure of a one parameter rational Bézier motion

Rational Bézier motion is one of the most fundamental type of motion. Here we discuss the trajectory of a point undergoing rational Bézier motion and affine control structure of motion. In dual quaternion representation , rigid transformation of a point is given by the following equation which is obtained by recasting Eq.(2.3) in terms of dual quaternions and the homogeneous coordinates of a point $\mathbf{P} : (P_1, P_2, p_3, P_4)$ of the object. Refer to Purwar and Ge [46] and Ge and Sirchia [75] for more details.

$$\widetilde{\mathbf{P}} = \mathbf{Q}\mathbf{P}\mathbf{Q}^* + P_4[(\mathbf{Q}^0)\mathbf{Q}^* - \mathbf{Q}(\mathbf{Q}^0)^*] \tag{2.13}$$

where, $\mathbf{Q}^*$ and $(\mathbf{Q}^0)^*$ are conjugates of $\mathbf{Q}$ and $\mathbf{Q}^0$, respectively, and $\widetilde{\mathbf{P}}$ denotes homogeneous coordinates of the point after displacement.

In the space of dual quaternions, for a given set of dual quaternions $\hat{\mathbf{Q}}_i$,

the following rational Bézier representation

$$\hat{\mathbf{Q}}(t) = \sum_{i=0}^{n} B_i^n(t)\hat{\mathbf{Q}}_i \tag{2.14}$$

defines a Bézier curve in the space of dual quaternions. where, $\hat{\mathbf{Q}}$ is a homogeneous dual quaternion and $B_i^n(t)$ are the Bernstein polynomials.The Bézier dual quaternion curve corresponds to a rational Bézier motion whose point trajectories are rational Bézier curves.

A representation for rational Bézier motion in the cartesian space can be obtained by substituting Eq. (2.14) in Eq. (2.13). The trajectory of a point undergoing rational Bézier motion obtained by above substitution is given by

$$\widetilde{\mathbf{P}}^{2n}(t) = [H^{2n}(t)]\mathbf{P} \tag{2.15}$$

$$[H^{2n}(t)] = \sum_{k=0}^{2n} B_k^{2n}(t)[H_k] \tag{2.16}$$

where $[H^{2n}(t)]$ is the matrix representation of the rational Bézier motion of degree 2n in cartesian space. The following matrices $[H_k]$, referred as Bézier control matrices define the affine control structure of motion. Refer Jütler and Wagner [43].

$$[H_k] = \frac{1}{C_k^{2n}} \sum_{i+j=k} C_i^n C_j^n w_i w_j [H_{ij}^*] \tag{2.17}$$

where,

$$[H_{ij}^*] = [H_i^+][H_j^-] + [H_j^-][H_i^{0+}] - [H_i^+][H_j^{0-}] + (\alpha_i - \alpha_j)[H_j^-][Q_i^+] \tag{2.18}$$

In the above equations, C_i^n and C_j^n are binomial coefficients and $\alpha_i = w_i^0/w_i$, $\alpha_i = w_j^0/w_j$ are the weight ratios and

$$[H_j^-] = \begin{bmatrix} q_{j,4} & -q_{j,3} & q_{j,2} & -q_{j,1} \\ q_{j,3} & q_{j,4} & -q_{j,1} & -q_{j,2} \\ -q_{j,2} & q_{j,1} & q_{j,4} & -q_{j,3} \\ q_{j,1} & q_{j,2} & q_{j,3} & q_{j,4} \end{bmatrix} \tag{2.19}$$

$$[Q_i^+] = \begin{bmatrix} 0 & 0 & 0 & q_{i,1} \\ 0 & 0 & 0 & q_{i,2} \\ 0 & 0 & 0 & q_{i,3} \\ 0 & 0 & 0 & q_{i,4} \end{bmatrix} \tag{2.20}$$

$$[H_i^{0+}] = \begin{bmatrix} 0 & 0 & 0 & q_{i,1}^0 \\ 0 & 0 & 0 & q_{i,2}^1 \\ 0 & 0 & 0 & q_{i,3}^2 \\ 0 & 0 & 0 & q_{i,4}^3 \end{bmatrix} \tag{2.21}$$

$$[H_j^{0-}] = \begin{bmatrix} 0 & 0 & 0 & -q_{j,1}^0 \\ 0 & 0 & 0 & -q_{j,2}^1 \\ 0 & 0 & 0 & -q_{j,3}^2 \\ 0 & 0 & 0 & -q_{j,4}^3 \end{bmatrix} \tag{2.22}$$

$$[H_i^+] = \begin{bmatrix} q_{i,4} & -q_{i,3} & q_{i,2} & q_{i,1} \\ q_{i,3} & q_{i,4} & -q_{i,1} & q_{i,2} \\ -q_{i,2} & q_{i,1} & q_{i,4} & q_{i,3} \\ -q_{i,1} & -q_{i,2} & -q_{i,3} & q_{i,4} \end{bmatrix} \tag{2.23}$$

In the above matrices, $(q_{i,1}, q_{i,2}, q_{i,3}, q_{i,4})$ are the four components of the real part of $(\mathbf{q}_i)$ and $(q_{i,1}^0, q_{i,2}^0, q_{i,3}^0, q_{i,4}^0)$ are the four components of the dual part of $(\mathbf{q}_i^0)$ of the unit dual quaternion $(\hat{\mathbf{q}}_i)$.

Chapter 3

Implemented Motion Design Algorithms

This chapter discusses the motion design schemes implemented so far in this motion design library (abbreviated as MDL). In this chapter basically algorithms for three major categories of motion design problems are discussed. First, the implementation of fundamental motion types which includes rational screw motion, rational Bézier motion,and rational B-spline motion is explained. Second, motion fitting algorithms, under which global interpolation to position data and least square motion approximation are elaborated. Lastly, a subdivision based motion design schemes which includes B-spline tweak, 4-point tweak and Jarek's tweak based on paper on Education driven CAD by Rossignac (See [48]) is discussed.

3.1 Discussion on implemented algorithms

3.1.1 Basic Motion Types

This section discusses the implementation of basic motion types such as rational screw motion, rational Bézier motion and rational B-spline motion. Effect of dual weights on this motion types is also discussed.

3.1.1.1 Rational Screw Motion

A screw motion can be considered as a simplest motion. It is a special combination of rotation and translation performed along the same axis, called screw axis. It can be used to represent displacement of an object from one position to another in 3-dimensional space which involves both translation and rotation. It is basically a linear interpolation in dual quaternion space, which corresponds to a rational screw motion in Euclidean three-space (E^3). It is given by

$$\hat{\mathbf{Q}}(t) = (1-t)\hat{w}_0\hat{\mathbf{q_0}} + t\hat{w}_1\hat{\mathbf{q_1}} \tag{3.1}$$

It has been shown by Ge and Ravani [30] that the above equation represents a quadratic rational motion with fixed screw axis and varying angular speed and pitch. Figure 3.1 shows a piece wise rational screw motion interpolating through five succesive control positions.

Effect of dual weights on rational motions is studied by Purwar and Ge [46]. The study shows that change of real part of dual weights leaves the trajectory of the screw motion invariant. However, it affects the parametrization of the motion. Figure 3.2(a) shows a screw motion with unit real weights and Fig-

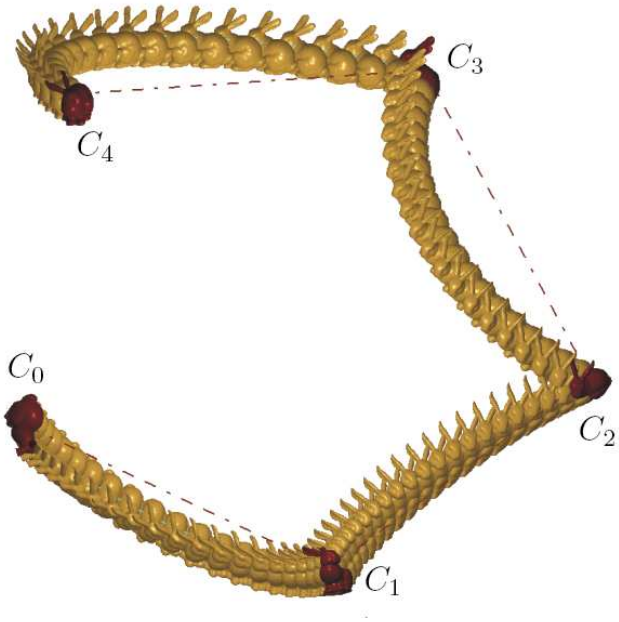


Figure 3.1: Piece-wise rational screw motion interpolating through five control positions $\mathbf{C}_i, (i = 0, ..., 4)$. All dual weights are set to unity. The dotted straight lines connects the successive control positions.

(a) (b)

Figure 3.2: (a)Screw motion between two control positions with real weights set to unity $\hat{w}_i = 1 + \epsilon 0; i = 0, 1$. (b) Screw motion between same control positions but with with non-unit real weights $\hat{w}_0 = 1 + \epsilon 0, \hat{w}_1 = 3 + \epsilon 0$.

ure 3.2(b) shows screw motion with same control position but with a different set of real weights. It is evident that the two screw motions have same path but different speed (or parametrization). Higher value of real weight for second control position results in slowing down of the object as it approaches the second position. Control positions are marked as $\mathbf{C}_i, (i = 0, ..., n)$. The coordinates of the given set of control positions using dual quaternion representation are listed in Table 3.1

C_i	q_1	q_2	q_3	q_4	q_1^0	q_2^0	q_3^0	q_4^0
C_0	0.6827	0.3413	0.4096	0.4996	-0.2649	1.0526	-1.1435	0.5803
C_1	0.0000	-0.3442	0.0000	0.9396	0.8758	0.7047	0.2132	0.2561

Table 3.1: Dual quaternion representation of given set of control positions for rational screw motion depicted in Fig. 3.2.

Now let us consider the effect of dual part of weight on rational screw motion. It has been shown by Purwar and Ge [46] that change in dual part of weight results in different translation at a particular value of t, while rotation components are same. It should be noted that even with non unit dual part of weight, motion still passes through end positions. It can be explained as

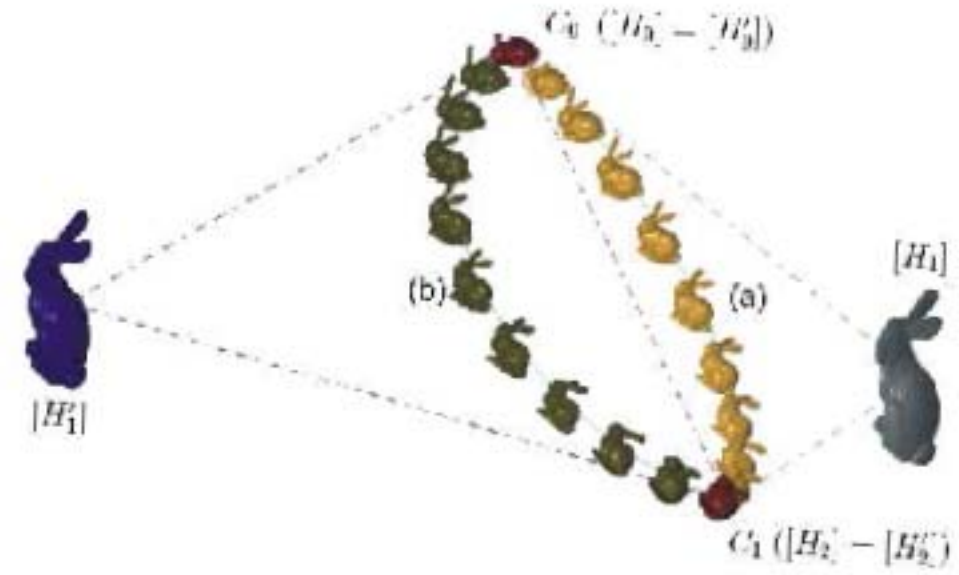


Figure 3.3: (a) Screw motion between two control positions with dual weights set to unity $\hat{w}_i = 1 + \epsilon 0; i = 0, 1$. (b) Screw motion between same control positions but with with non-zero dual part of weights $\hat{w}_0 = 1 + \epsilon 0, \hat{w}_1 = 1 + \epsilon 3$.

follows. Let $\hat{w_0} = 1 + \epsilon 0$; $\hat{w_1} = 1 + \epsilon 3$. Then we have

$$\hat{\mathbf{Q}}(t) = (1 - t)\hat{\mathbf{q_0}} + t(1 + \epsilon 3)\hat{\mathbf{q_1}} \tag{3.2}$$

or,

$$\hat{\mathbf{Q}}(t) = (1 - t)\hat{\mathbf{q_0}} + t\hat{\mathbf{q_1}} + \epsilon 3t\hat{\mathbf{q_1}} \tag{3.3}$$

It can be easily seen that at $t = 0$, $\hat{\mathbf{Q}}(t) = \hat{\mathbf{q_0}}$, and at $t = 1$, $\hat{\mathbf{Q}}(t) = (1 + \epsilon 3)\hat{\mathbf{q_1}}$, which represents same spatial displacement as $\hat{\mathbf{q_1}}$, which is a unit dual quaternion. ‘

From Figure 3.3, we can see that, since real part of the weights does not change, the orientation of objects at same instant (for same value of t) does

not change. In terms of matrix representation of the trajectory of motion, it can be shown that the dual part of the weight translates the middle control matrix from $[H_1]$ to $[H_1']$ and thus changes the path of screw motion.

3.1.1.2 Rational Bézier Motion

The implementation to plot a rational Bézier motion is largely based on paper by Purwar and Ge [46] which discuss the effect of dual weights on rational motions. In the space of dual quaternions, for a given set of unit dual quaternions and dual weights $\hat{\mathbf{q}}_i$, $\hat{w}_i$; $(i = 0, ..., n)$ respectively, a rational Bézier motion is given by

$$\hat{\mathbf{Q}}(t) = \sum_{i=0}^{n} B_i^n(t)\hat{\mathbf{Q}}_i = \sum_{i=0}^{n} B_i^n(t)\hat{w}_i\hat{\mathbf{q}}_i \tag{3.4}$$

where, $\hat{\mathbf{Q}}$ is a homogeneous dual quaternion given by $\hat{\mathbf{Q}} = \mathbf{Q} + \epsilon\mathbf{Q}^0$, where $\mathbf{Q} = w\mathbf{q}$, $\mathbf{Q}^0 = w\mathbf{q}^0 + w^0\mathbf{q}$. This is obtained by expanding $\hat{\mathbf{Q}} = \hat{w}\hat{\mathbf{q}}$ using dual number algebra. Here $\hat{\mathbf{q}} = \mathbf{q} + \epsilon\mathbf{q}^0$ represents a unit dual quaternion. $B_i^n(t)$ are the Bernstein polynomials. The Bézier dual quaternion curve given by Eq. (3.4) defines a rational Bézier motion of degree $2n$. Figure 3.4 shows a rational Bézier motion corresponding to a given set four control positions. Weights for all four control positions is set to unity. The coordinates of the given set of control positions using dual quaternion representation are listed in Table 3.2

Figure 3.5 shows a rational Bézier motion corresponding to a given set five control positions with its affine control structure.

The following part discusses the effect of weights and reparameterization of rational Bézier motion. First we will consider the effect of real part of dual

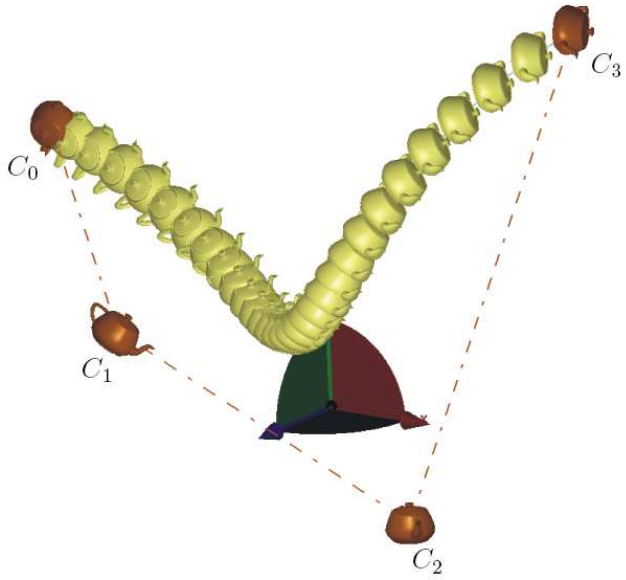


Figure 3.4: Rational Bézier motion of degree 6 corresponding to a given set of four control positions $\mathbf{C}_i, (i = 0, ..., 3)$. All dual weights are set to unity.

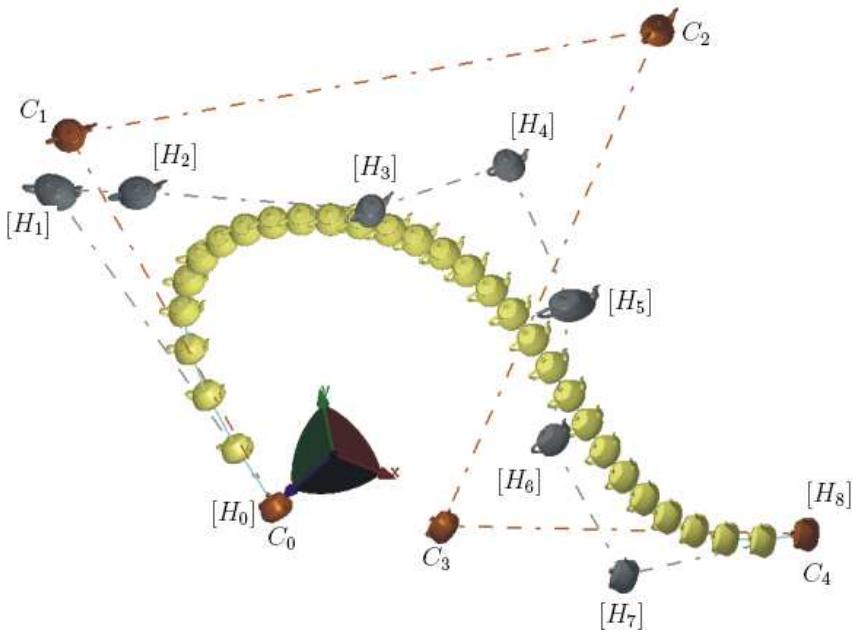


Figure 3.5: Rational Bézier motion corresponding to a given set of five control positions $(C_i; i = 0...4$ with its affine control structure $([H_i]; i = 0...8$.

C_i	q_1	q_2	q_3	q_4	q_1^0	q_2^0	q_3^0	q_4^0
C_0	0.6827	0.3413	0.4096	0.4996	-0.2649	1.0526	-1.1435	0.5803
C_1	-0.01478	0.0887	-0.2956	0.9393	-0.6693	0.1642	0.7269	0.1079
C_2	0.0000	-0.3442	0.0000	0.9396	0.8758	0.7047	0.2132	0.2561
C_3	-0.5109	-0.2554	-0.5109	0.6425	-0.4759	1.7301	0.0928	0.3831

Table 3.2: Dual quaternion representation of given set of control positions for rational screw motion depicted in Fig. 3.4.

weight. It has been shown by Purwar and Ge [46] changing real weights change the parametrization (speed) as well as path of the motion. However, there exists a reparameterization of rational Bézier motion that makes path of Bézier motion invariant. In CAGD, such a reparameterization corresponds to a rational linear parameter transformations (See Farin [6]; Chapter 13) of the following type:

$$t = \frac{t'}{\rho(1-t') + t'} \tag{3.5}$$

Patterson [43] has shown that such a projective transformation leaves the shape of rational curve invariant if each weight w_i is replaced by $\rho_{n-i}w_i$, where ρ is a non zero real number. Thus, if the real weights of a dual quaternion curve are transformed in a similar fashion, then shape of dual quaternion curve remains invariant. Consider the following weight transformations: $\hat{\nu}_i = \lambda^{n-i}\hat{w}_i$, where, λ is any non zero real scaler and $\hat{\nu}_i$ can be either real or dual weights. On expanding we get,

$$\nu_i = \lambda^{n-i}w_i \tag{3.6}$$

$$\nu_i^0 = \lambda^{n-i} w_i^0 \tag{3.7}$$

where, ν_i and ν_i^0 are real and dual part os the new weight $\hat{\nu}_i$.

On substituting Eq. (3.6) and Eq. (3.7) in Eq. (3.9), we get control matrix $[\tilde{H_k}]$

$$[\tilde{H_k}] = \sum_{i+j=k} \frac{{}^nC_i^n C_j}{{}^{2n}C_k} \lambda^{2n-(i+j)} w_i w_j [H_{ij}^*] = \lambda^{2n-k} [H_k] \tag{3.8}$$

The trajectory of a point is given by

$$[\tilde{\mathbf{P}}^{2n}(t)] = \sum_{k=0}^{2n} B_k^{2n}(t) \lambda^{2n-k} [H_k] \mathbf{P} \tag{3.9}$$

The above equation shows that transforming the weights via Eq. (3.6) and Eq. (3.7) is equivalent to multiplying the Bézier control positions ($[H_k]$) of the motion by a scaler λ^{2n-k}. This means that the path of the trajectory of any point $\mathbf{P}$ under the transformed motion is invariant. It follows that the weight change as defined by Eq. (3.6) and Eq. (3.7) does not change the path of the motion. However, the speed of the resulting motion is in general different from the original motion.

As an example, consider a rational Bézier motion of degree 6 as defined by a cubic dual quaternion Bézier curve:

$$\tilde{\mathbf{Q}}(t) = \sum_{i=0}^{3} B_i^3(t) \hat{w}_i \hat{\mathbf{q}}_\mathbf{i} \tag{3.10}$$

Figure 3.6(a) shows this motion for a given set of four control positions marked by C_i. Each control position has unit real weight associated with

them. Figure 3.6(a) also shows affine control structure in grey color indicated by $[H_i]$. Figure 3.6(b) shows degree six rational Bézier motion for the same set of control position but with different real weights associated with them. It is evident that changes in real weights of control positions results in a changed affine control structure as defined by control matrices $[H_i]$, and hence results in a change in path of the resulting motion.

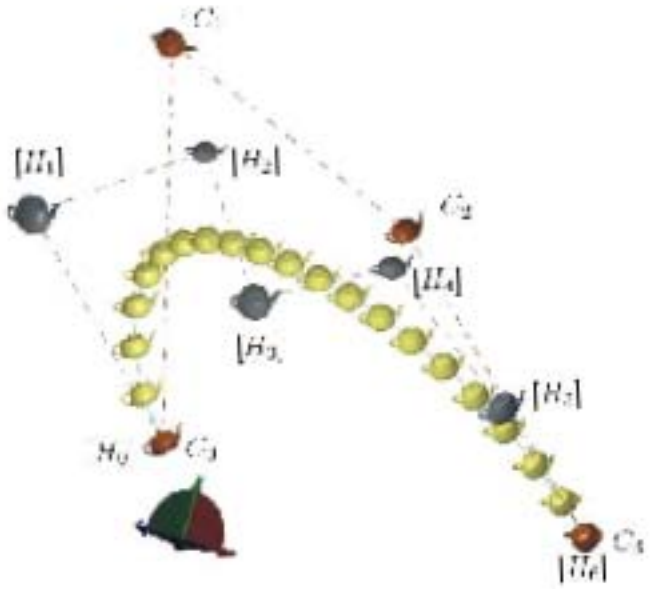
Figure 3.7(a) and Figure 3.7 illustrate a rational Bézier motion of degree six prior to reparameterization and after applying reparameterization respectively as explained before. It demonstrates that when weights are scaled as per Eq. (3.6) and Eq. (3.7), the path of the motion does not change, although the parameterization (or speed) along the path does change.

In the next part we will see the effect of dual part of weight on the rational Bézier motion. In general, an n^{th} degree Bézier curve in the space of dual quaternion is given by

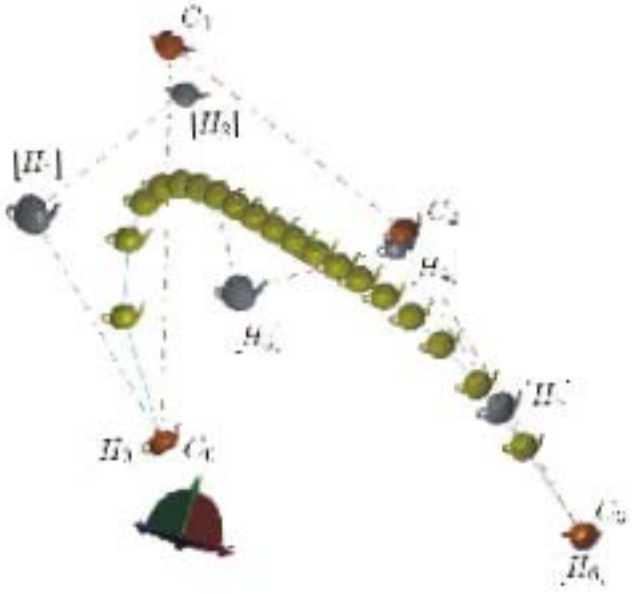
$$\tilde{\mathbf{Q}}(t) = \sum_{i=0}^{n} B_i^n(t)\hat{w}_i\hat{\mathbf{q}}_\mathbf{i} \tag{3.11}$$

Let, for a particular k, $\hat{w}_k = w_k + \epsilon w_k^0$, where we impose the condition that $w_k^0 \neq 0$, i.e., we assume that there is only one dual weight $\hat{w}_k$ that has a non zero dual part. The intent is to separate the effect of in just the dual part and the effect is achieved by restricting ourselves to just one weight. Hence, $w_i^0 = 0$ for all $i \neq k$ .Then we get,

$$\tilde{\mathbf{Q}}(t) = \sum_{i=0}^{n} B_i^0(t)w_i\hat{\mathbf{q}}_\mathbf{i} + \epsilon B_k^n(t)w_k^0\mathbf{q}_k \tag{3.12}$$

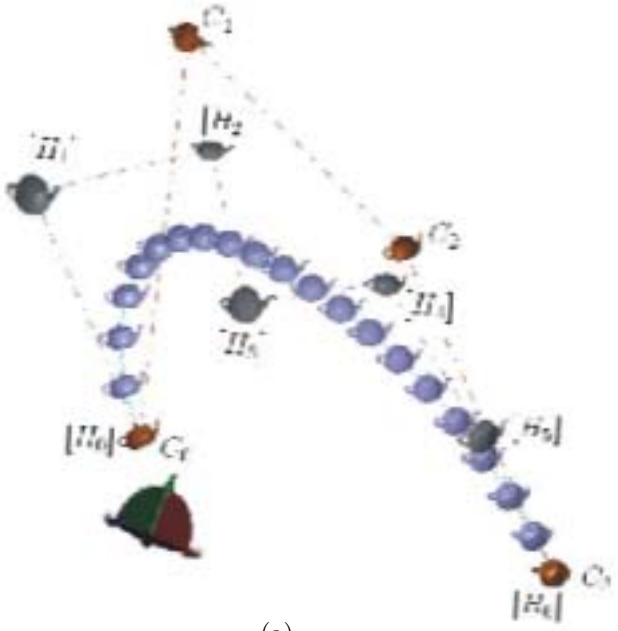


(a)

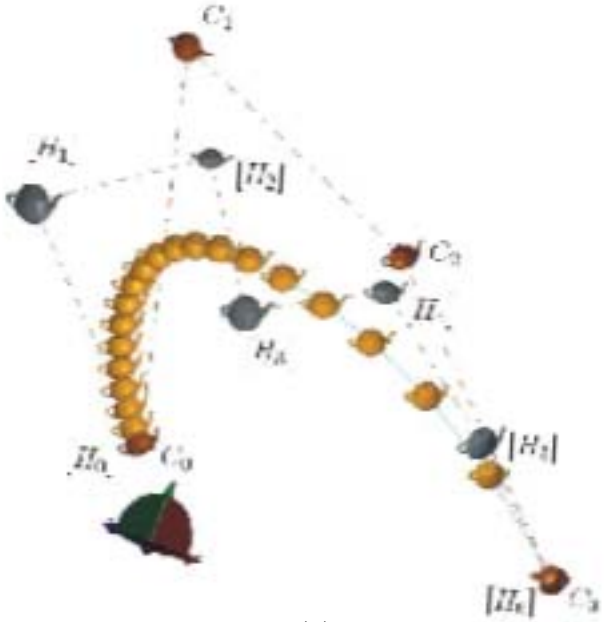


(b)

Figure 3.6: (a) A rational Bézier motion of degree six with unit real weights, $\hat{w}_i = 1 + \epsilon 0; i = 0, ..., 3$. (b) A rational Bézier motion with non unit real weights $\hat{w}_i = 1 + \epsilon 0; i = 0, 3$ and $\hat{w}_i = 3 + \epsilon 0; i = 1, 2$.



(a)



(b)

Figure 3.7: (a) A rational Bézier motion of degree six with weights, $\hat{\mathbf{w}}_i = 1 + \epsilon 1; i = 0, ..., 3$. (b) A reparameterized rational Bézier motion with weights $\hat{w}_i = \lambda^i + \epsilon\lambda^i; \lambda = 2$ and $i = 0, ..., 3$.

Based on above equation, a new translation vector $\mathbf{d}'$ can be found from Eq. (2.7)as

$$\mathbf{d}'(t) = \frac{(\mathbf{Q^0})\mathbf{Q}^* - \mathbf{Q}(\mathbf{Q^0})^*}{\mathbf{Q}\mathbf{Q}^*} + \Delta d(t) \tag{3.13}$$

where,

$$\Delta d(t) = \frac{w_k^0 B_k^n(t)(\mathbf{q}_k\mathbf{Q}^* - \mathbf{Q}\mathbf{q}_k^*)}{\mathbf{Q}\mathbf{Q}^*} \tag{3.14}$$

and,

$$\mathbf{Q}(t) = \sum_{i=0}^{n} B_i^n(t) w_i \mathbf{q_i} \tag{3.15}$$

$$\mathbf{Q}^0(t) = \sum_{i=0}^{n} B_i^n(t) w_i \mathbf{q_i^0} \tag{3.16}$$

Eq. (3.13) implies that the introduction of the non zero part of the dual weight (w_k^0) adds a translation component Δd to the rational Bézier motion, while the rotation component remains unaltered.

In general, if there are more than one weight, which have non zero dual parts, $\Delta d(t)$ in Eq. (3.14) changes accordingly to:

$$\Delta d(t) = \frac{\sum_{i=k(k\geq 0)}^{m(m\leq n)} w_i^0 B_i^n(t)(\mathbf{q}_i\mathbf{Q}^* - \mathbf{Q}\mathbf{q}_k^*)}{\mathbf{Q}\mathbf{Q}^*}; w_i^0 = 0 \text{ if } i \in [k, m] \tag{3.17}$$

where, (m-k)+1 are the number of consecutive weights with a non zero dual part. In particular, since at $t = 0$, we have $B_i^n(t)$ for $i \neq 0$, it follows that

$$\Delta d(t) = \frac{w_0^0[\mathbf{q}_0(w_0\mathbf{q})^* - (w_0\mathbf{q}_0)\mathbf{q}_0^*]}{w_0 w_0 \mathbf{q_0}\mathbf{q_0}^*} = 0 \tag{3.18}$$

The same result hold true at $t = 1$. This indicates that even though introducing dual weights gives rise to an extra translation component, it does not

violate the end point interpolation property, as commonly and desirably found in Bézier curves. Since this extra translation component (Δd) is time dependent, its effect is to change the motion via translation that varies in direction and magnitude along the path of the motion. It is exemplified in Figure 3.8, where it can be seen that due to non zero dual part of weight the trajectories of the motion differ but for the same instant (at the same value of t) the orientation of each teapot matches with its counterpart with modified dual weight. Notice the poring mouth and handle of teapots at t=0.05 and at t=0.95. The control positions are labeled as C_i.

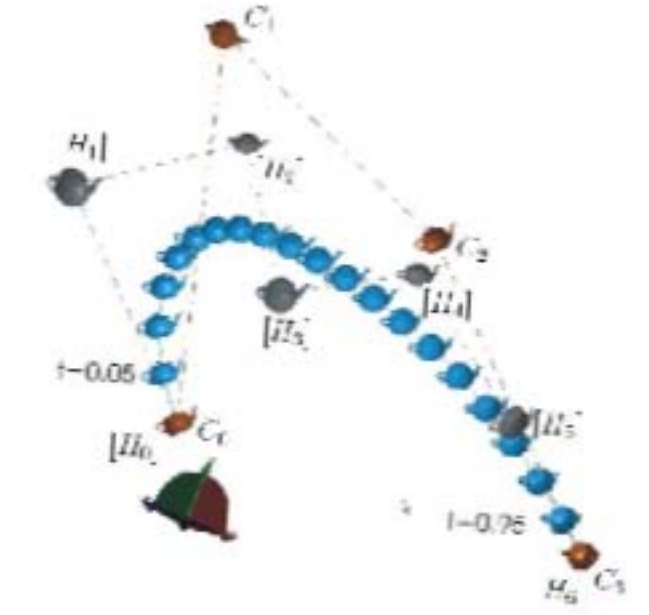
3.1.1.3 Rational B-Spline Motion

A rational B-Spline curve in dual quaternion space, which represents a NURBS motion of degree $2p$, is given by,

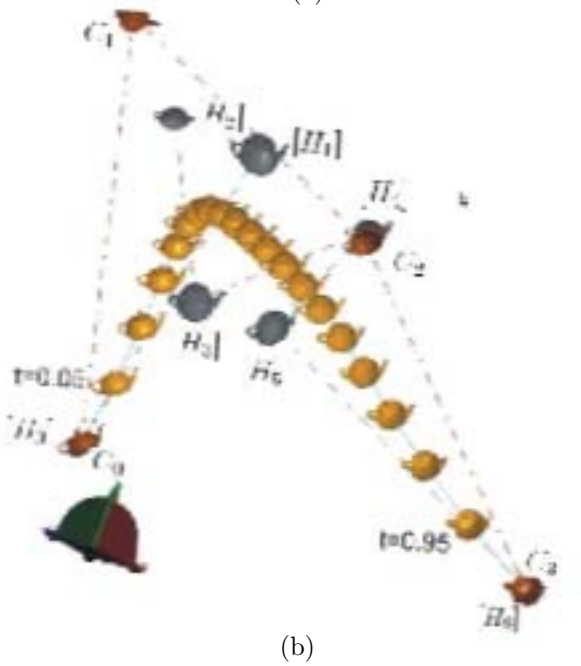
$$\hat{\mathbf{Q}}(t) = \sum_{i=0}^{n} N_{i,p}(t)\hat{\mathbf{Q}}_i = \sum_{i=0}^{n} N_{i,p}(t)\hat{w}_i\hat{\mathbf{q}}_i \tag{3.19}$$

where, $N_{i,p}$ are the p^th degree B-spline basis functions.

Fig. 3.9 shows a rational B-Spline for a given set of five control positions. The coordinates of the given set of control positions using dual quaternion representation along with the associated dual weights are listed in Table 3.3



(a)



(b)

Figure 3.8: (a) A rational Bézier motion of degree six with weights, $\hat{w}_i = 1 + \epsilon 0; i = 0, ..., 3$. (b) Effect of dual weights: A rational Bézier motion with weights $\hat{w}_i = 1 + \epsilon 0; i = 0, 3$ and $\hat{w}_i = 1 + \epsilon 4; i = 1, 2$

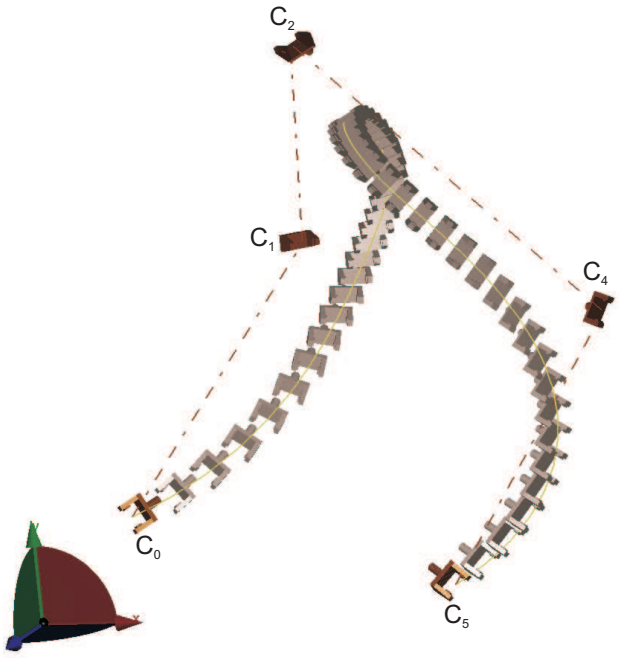


Figure 3.9: Rational B-spline motion corresponding to a given set of five control positions marked as $\mathbf{C}_i, (i = 0, ..., 5)$.

C_i	$\hat{w}_i$	q_1	q_2	q_3	q_4	q_1^0	q_2^0	q_3^0	q_4^0
C_0	$1+\epsilon 0$	0.3314	0.1913	-0.4619	0.8001	0.1691	0.6310	-0.2614	-0.0700
C_1	$1+\epsilon 0$	0.0000	0.3827	0.0000	0.9239	1.4022	1.3858	-0.7721	-0.5740
C_2	$1+\epsilon 0$	-2.4999	-0.0669	0.2499	0.9330	2.6998	1.9865	1.7283	0.3885
C_3	$1+\epsilon 0$	0.1913	0.3314	-0.4619	0.8001	3.1116	0.0704	2.0253	-1.9426
C_4	$1+\epsilon 0$	0.4619	0.4619	-0.1913	0.8446	1.7172	0.0116	1.3462	-1.2505

Table 3.3: Dual quaternion representation of a given set of control positions for rational B-Spline motion of degree 8 depicted in Fig. 3.9.

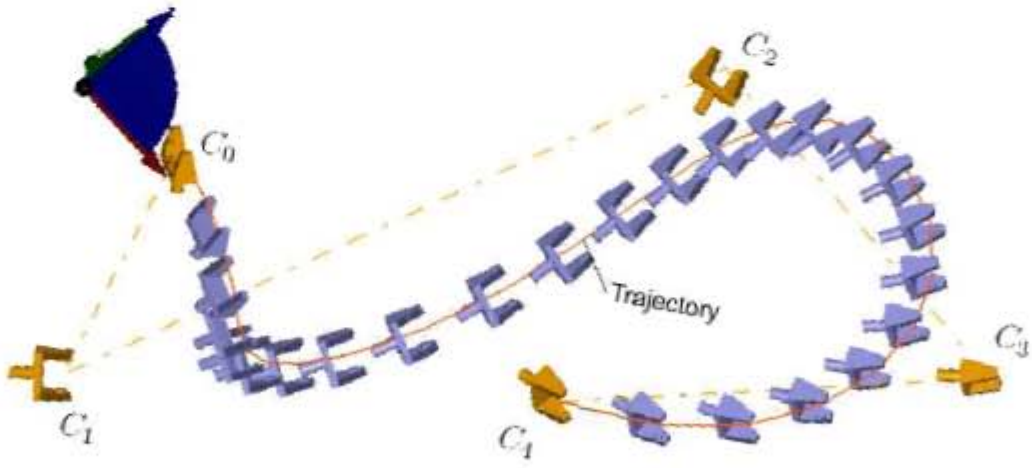
Rational B-spline motion has a piecewise rational Bézier form, hence the effects weights on a rational B-spline motion is similar to that of rational Bézier motion. Fig 3.10(a) shows five control positions (labeled as C_i) and the trajectory followed by the object undergoing a rational B-spline motion of degree 4.

It should be noted that it corresponds to a dual quaternion curve of degree 2. Fig. 3.10(b) shows the rational B-spline motion for same control positions for a different choice of weights. Observe the change in parametrization (speed) near C_2 The control positions (C_i) are same as listed in Table 3.3.

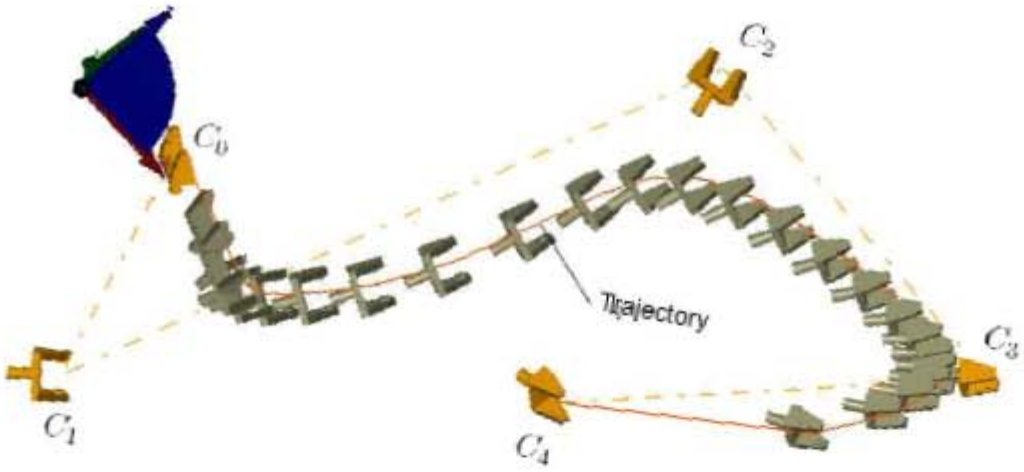
3.1.2 Motion Fitting

In motion fitting the focus is on designing of NURBS motion which fit a rather arbitrary set of geometric data such as control position and derivative vectors. We can divide motion fitting problem into two basic categories namely, motion fitting and motion approximation. In motion interpolation we design a motion which passes through given data precisely, for example the motion passing through given control positions and assumes that derivatives at the prescribed control positions. In motion approximation, as the name suggests we construct motion which do not necessarily satisfy the given data precisely, but only approximately. In motion approximation problem it is often desirable to specify a maximum bound on the derivation of the motion from the given data, and to specify certain constraints, i.e, data which is to be satisfied precisely.

For a typical motion fitting problem, input is generally geometric data such as control positions and derivatives. Output is NURBS motion. Furthermore, either the degree p must be input to the algorithm or algorithm must select an appropriate degree. If C^r continuity is desired for a dual quaternion curve,



(a)



(b)

Figure 3.10: (a) Rational B-spline motion corresponding to a given set of five control positions of degree 4, $\hat{w}_i = 1 + \epsilon 0; i = 0, ..., 3$. (b) Effect of weights: A rational B-spline motion with weights $\hat{w}_i = 1 + \epsilon 0; i = 0, 1, 4$ and $\hat{w}_2 = 1 + \epsilon 5, \hat{w}_3 = 1 + \epsilon 5$.

then the chosen degree p must satisfy

$$p \geq r + 1 \tag{3.20}$$

(assuming no interior knots of multiplicity > 1). Assuming no other requirements, choosing $p = r + 1$ is usually adequate for interpolation. For approximation, choosing $p > r + 1$ may produce better results. See Piegel and Tiller [10]

There are thousands of NURBS curves which can interpolate or approximate a given data set. However, in this thesis, I have restricted myself to two motion fitting algorithms namely, Global motion interpolation to position data and Least square motion approximation. As the name suggests, the first one falls under the category of motion interpolation, and the second one falls under the category of motion approximation. They are discussed in detail in following sections.

3.1.2.1 Global Motion Interpolation to Position Data

Implementation of this algorithm is extension of Global Curve Interpolation to Point Data algorithm described in Piegl and Tiller [10] to dual quaternion space. The curve designed in dual quaternion space is then projected back to E^3 to give a global motion interpolation which interpolates through a given set of input positions. Suppose we are given a set of positions $\mathbf{C}_k, k = 0, ..., n$. We want to interpolate this motion with a pth degree nonrational B-spline dual quaternion curve. If we assign a parameter value $\bar{u}_k$ to each C_k, and select an appropriate knot vector $U = u_0, ..., u_m$, we can set up the $(n + 1) * (n + 1)$

system of linear equations

$$\mathbf{C}_k = \sum_{i=0}^{n} N_{i,p}(\bar{u}_k)\mathbf{P}_i \tag{3.21}$$

The control positions, $\mathbf{P}_i$, are the $n+1$ unknowns. Let r be the number of coordinates in C_k. In dual quaternion representation r is 8 i.e., $(q_1, q_2, q_3, q_4, q_1^0 q_1^0 q_2^0 q_3^0 q_4^0)$. It should be noted that this method is independent of r. Eq. (3.21) has one coefficient matrix, with r right hand sides and, correspondingly, r solution sets for the r coordinates of the P_i.

The problem choosing the parameter $\bar{u}_k$ and knot vector U remains and their choice affects the design and parametrization of motion. It has been assumed that the parameter $\bar{u}_k$ lies in the range $u \in [0, 1]$. Three common methods of choosing u_k are equally spaced, chord length and centripetal method.

Knots can be equally spaced or can be defined by using method of averaging. This methods are described in Piegl and Tiller [10]. In the current implementation parameter u_k as well as knot vector is equally spaced. However, this can produce erratic shapes when data is unevenly spaced. Parameter u_k is computed as per following equation:

$$\begin{aligned} &\bar{u}_0 = 0 \ ; \ \bar{u}_n = 1 \\ &\bar{u}_k = \tfrac{k}{n} \ , \ k = 1, ..., n-1 \end{aligned} \tag{3.22}$$

The equally spaced knot vector is computed as follows

$$\begin{aligned} &u_0 = ... = u_p = 0 \ ; \ u_{m-p} = ... = u_m = 1 \\ &\bar{u}_{j+p} = \tfrac{j}{n-p+1} \ , \ j = 1, ..., n-p \end{aligned} \tag{3.23}$$

Once the parameter u_k and the knots are computed, the $(n+1)*(n+1)$ coefficient matrix of the system (Eq. (3.21)) is set up by evaluating the non-zero basis functions at each $u_k, k = 0, .., n$. The control positions that we get by solving Eq. (3.21) is then used as input to to B-spline motion design algorithm to plot Global motion interpolation. Fig. 3.11 shows a global motion interpolation for a given set of five positions. The key frames shown in green color are the computed control positions(P_i). The coordinates of the given set of input positions (C_i)(shown in red color) using dual quaternion representation are listed in Table 3.4.

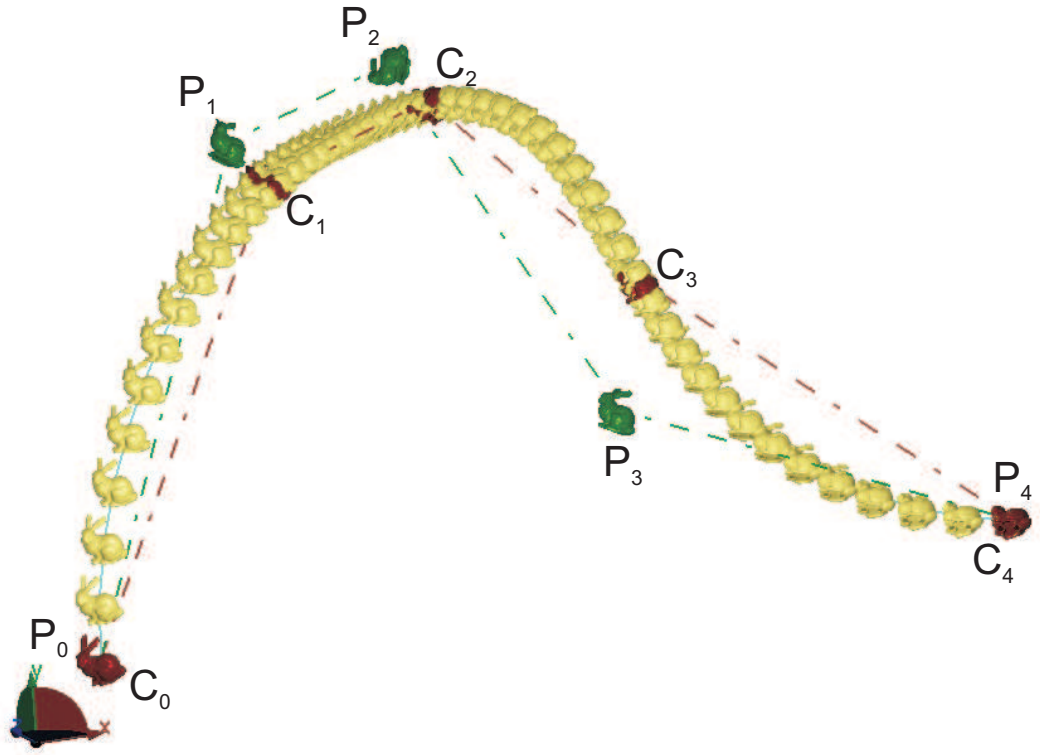


Figure 3.11: Global motion interpolation interpolating through given set of five input positions$\mathbf{C}_i, (i = 0, ..., 4)$. The degree of the motion is 4. The computed control positions are marked as $\mathbf{P}_i, (i = 0, ..., 4)$.

The control positions (C_i) are listed in Table 3.4.

C_i	q_1	q_2	q_3	q_4	q_1^0	q_2^0	q_3^0	q_4^0
C_0	0.3827	0.0000	0.0000	0.9239	0.4619	0.4619	0.1913	-0.1913
C_1	0.0000	0.2588	0.0000	0.9659	1.8672	3.3808	0.7591	-0.9059
C_2	0.0000	0.0000	0.3827	0.9239	4.3024	2.5475	-0.2309	0.0957
C_3	0.0000	0.0000	0.0000	1.0000	4.5000	2.5000	0.0000	0.0000
C_4	-0.3827	0.0000	0.0000	0.9238	6.4672	0.46194	0.1913	2.6788

Table 3.4: Dual quaternion representation of given set of input positions for global motion interpolation of degree 8 depicted in Fig. 3.11.

3.1.2.2 Least Square Approximation

Least square motion approximation is based on extension of least square curve approximation described in Piegl and Tiller [10] to dual quaternion space. The curve designed in dual quaternion space is then projected back to E^3 to give a least square motion approximation which approximates through a given set of input positions. Let us assume that degree $p > 1, n \geq p$ and input positions $\mathbf{C_0}, ..., \mathbf{C_m}, (m > n)$ are given. The parameters $\bar{u}_k$ can be evaluated in a similar fashion as mentioned in description global motion interpolation to position data. In the current implementation of least square motion approximation Eq. (3.22) is used to compute $\bar{u}_k$. The equally spaced Knot Vector is computed using Eq. (3.23). We then set up and solve the (unique) linear least square problem for unknown control positions $\mathbf{P}_i$. We seek a degree p non rational curve

$$\mathbf{C}(u) = \sum_{i=0}^{n} N_{i,p}(u)\mathbf{P}_i \ ; \ u \in [0,1] \tag{3.24}$$

satisfying that

$$\begin{array}{l} \mathbf{C}_0 = \mathbf{C}(0) \text{ and } \mathbf{C}_m = \mathbf{C}(1) \\ \text{The remaining } \mathbf{C}_k \text{ are computed in the least square sense, i.e.,} \\ \sum_{k=1}^{m-1} |\mathbf{C}_k - \mathbf{C}(\bar{u}_k)|^2 \end{array} \tag{3.25}$$

is minimum with respect to $n+1$ variables, $\mathbf{P}_i$. the resulting motion generally does not precisely pass through $\mathbf{C_k}$, and $\mathbf{C}(\bar{u}_k)$ is not the closest position on $\mathbf{C}_k$ on $\mathbf{C}(u)$.

Fig. 3.12 shows a least square motion approximation for a given set of five positions. The key frames shown in green color are the computed control

positions(P_i). The coordinates of the given set of input positions (C_k)(shown in red color) using dual quaternion representation are listed in Table 3.5

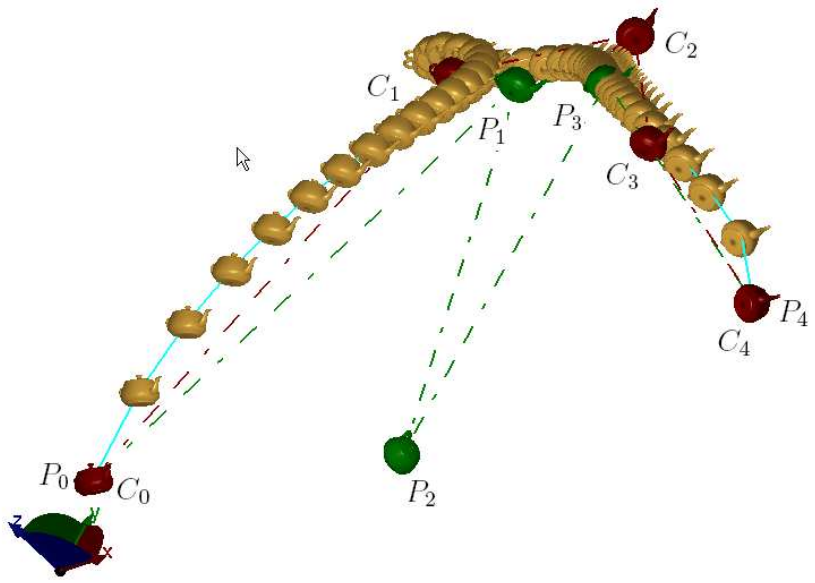


Figure 3.12: Motion approximating a set of five input positions $\mathbf{C}_i, (i = 0, ..., 4)$ based on least square motion approximation. The degree of the motion is 4. The computed control positions are marked as $\mathbf{P}_i, (i = 0, ..., 4)$.

The control positions (C_k) are as listed in Table 3.5.

C_i	q_1	q_2	q_3	q_4	q_1^0	q_2^0	q_3^0	q_4^0
C_0	0.3827	0.0000	0.0000	0.9239	0.4619	0.4619	0.1913	-0.1913
C_1	0.0000	0.2588	0.0000	0.9659	1.8672	3.3808	0.7591	-0.9059
C_2	0.0000	0.0000	0.3827	0.9239	4.6850	3.4714	-0.2309	0.0957
C_3	0.0000	0.0000	0.0000	1.0000	4.5000	2.5000	0.0000	0.0000
C_4	-0.3827	0.0000	0.0000	0.9238	6.4672	0.46194	01913	2.6788

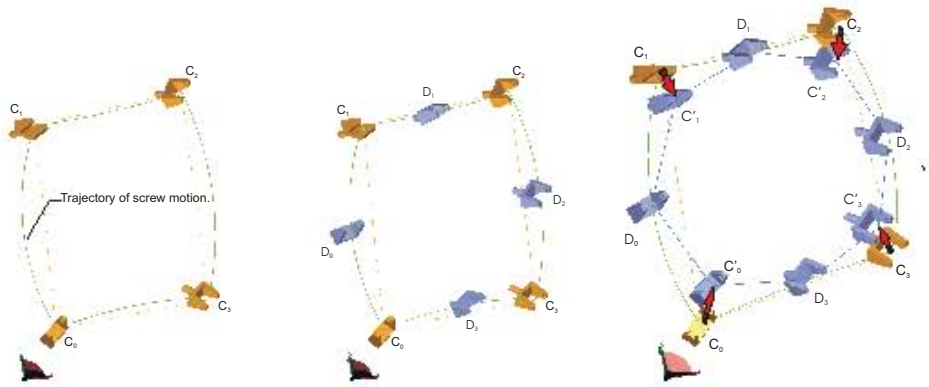
Table 3.5: Dual quaternion representation of given set of input positions for least square motion approximation of degree 4 depicted in Fig. 3.12.

3.1.3 Subdivision Motion

Subdivision motion implementations discussed in this thesis are based on paper on Education-Drive Research in CAD by Jarek Rossignac [48]. I have extended the simple subdivision processes that generate uniform cubic B-splines and 4-point subdivision curves to dual quaternion space to generate uniform cubic B-splines and 4-point subdivision dual quaternion curve. Initially we consider a closed loop control structure in E^3. Then this control structure is refined using split and tweak process. In this process, first, each edge of control structure in dual quaternion space (which corresponds to a rational screw motion in E^3). Then tweak either the new vertices, or the old ones, or both. Repeat this process as desired. After each split and tweak refinement, the number of positions in the control structure has doubled. A good tweak rule will increase the smoothness of the control structure with each refinement. Three different types of subdivision motions namely B-spline Tweak, 4-point tweak and Jarek's Tweak are elaborated in the following sections.

3.1.3.1 B-Spline Tweak

In B-spline tweak scheme for motion design, old positions are moved half way towards the average of their new neighbors. New neighbors (D_j) are generated by splitting each edge of closed control structure in dual quaternion space into half. They basically corresponds to a position at $t = 0.5$ of rational screw motion between successive control positions of control structure in $E3$ i.e. D_0 corresponds to a position at $t = 0.5$ of rational screw motion between C_0



(a) Control Structure (b) Split (c) Tweak

Figure 3.13: The closed loop control structure with control positions C_i, $i = 0, ..., 3$ is subdivided in two steps. First (b), the split step inserts new control positions D_j, $j = 0, ..., 3$. Then (c), the B-spline tweak step adjusts the original vertices C_i by positioning them half way, towards the average of their new neighbors. The adjusted vertices are labeled C'_i.

amd C_1 and so on. Then the original control positions C_i are tweaked to new positions labeled as C'_i such that they are positioned half way towards the average of their new neighbors, for example $C'_1 = mid(C_1, mid(D_0, D_1))$, where function $mid(x, y)$ gives the position on a screw motion corresponding to $t = 0.5$ between positions x and y. Fig.3.13 shows this steps pictorially.

The control positions (C_i) are as listed in Table 3.6.

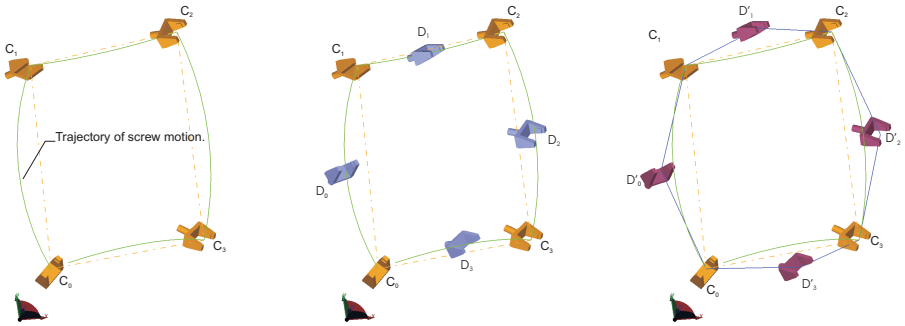
C_i	q_1	q_2	q_3	q_4	q_1^0	q_2^0	q_3^0	q_4^0
C_0	0.0000	0.0000	0.3827	0.9238	0.6533	0.2706	0.9238	0.3827
C_1	0.0000	0.3827	0.0000	0.9239	0.3034	0.6955	2.039	-1.5307
C_2	0.3079	0.3236	0.4267	0.8428	3.2645	3.0150	2.2678	-3.4981
C_4	0.5319	0.3919	0.2006	0.8223	3.1289	0.3259	0.7067	-2.3518

Table 3.6: Dual quaternion representation of given set of four input positions for various Subdivision motions

Repeated refinements that use a B-spline tweak procedure produces a motion that converges to a uniform cubic B-spline curve in quaternion space, which has apiece-wise polynomial formulation.

3.1.3.2 4-Point Tweak

In 4-point tweak scheme for motion design, new positions are moved by one quarter away from the average of their second degree neighbors. New positions (D_j) are generated by splitting each edge of closed control structure in dual quaternion space into half. They basically corresponds to a position at $t = 0.5$ of rational screw motion between successive control positions of control structure in $E3$ i.e. D_0 corresponds to a position at $t = 0.5$ of rational screw motion between C_0 and C_1 and so on. Then these newly generated positions D_i are tweaked to new positions labeled as D_i' such that they are moved one quarter away from the average of their second degree neighbors. For example $D_1' = (D_1 - mid(D_0, D_2))/4$, where function $mid(x, y)$ gives the position on a screw motion corresponding to $t = 0.5$ between positions x and y. The resulting motion interpolates the initial positions and bulges out Fig.3.14 shows this steps pictorially.



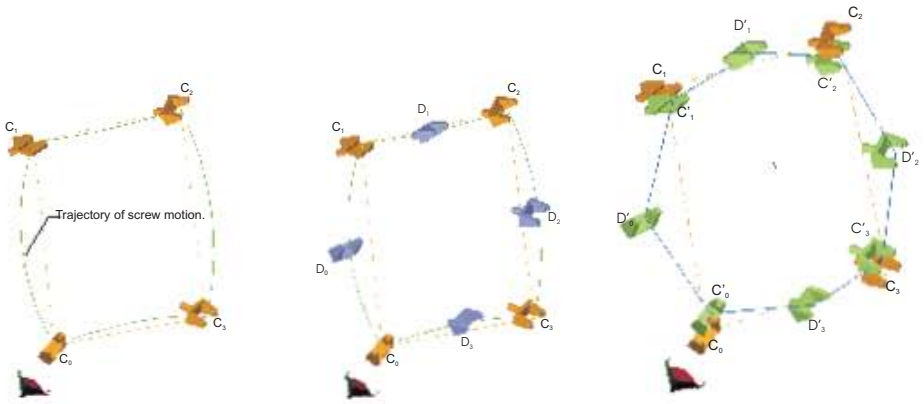
(a) Control Structure (b) Split (c) Tweak

Figure 3.14: The closed loop control structure with control positions C_i, $i = 0, ..., 3$ is subdivided in two steps. First (b), the split step inserts new control positions D_j, $j = 0, ..., 3$. Then (c), the 4-point tweak step adjusts the new positions D_j by positioning them by one quarter away from the average of their second degree neighbors. The adjusted vertices are labeled D'_j.

The control positions (C_i) are as listed in Table 3.6.

3.1.3.3 Jarek's Tweak

Jarek's tweak scheme of motion design generates a motion which lies in-between the B-spline and the 4-point tweak motions and is a closer approximation to the original control structure than either of these two. In this scheme old positions are moved by half of the B-spline tweak and the new positions by half of the 4-point tweak displacements. Fig.3.15 shows this steps pictorially.



(a) Control Structure (b) Split (c) Tweak

Figure 3.15: The closed loop control structure with control positions C_i, $i = 0, ..., 3$ is subdivided in two steps. First (b), the split step inserts new control positions D_j, $j = 0, ..., 3$. Then (c), the Jarek's tweak step adjusts the old positions C_i by half the displacement suggested by the B-spline tweak and the new positions D_j by half the displacement suggested by the 4-point tweak.

The control positions (C_i) are as listed in Table 3.6.

All three subdivision based motion design schemes are plotted simultaneously in Fig. 3.16. B-spline Tweak is shown in red, 4-point tweak is shown in green and Jarek's tweak is shown in blue. The common control structure for all three motions is shown in black. The control positions of the L-shaped polygon are as listed in Table 3.7.

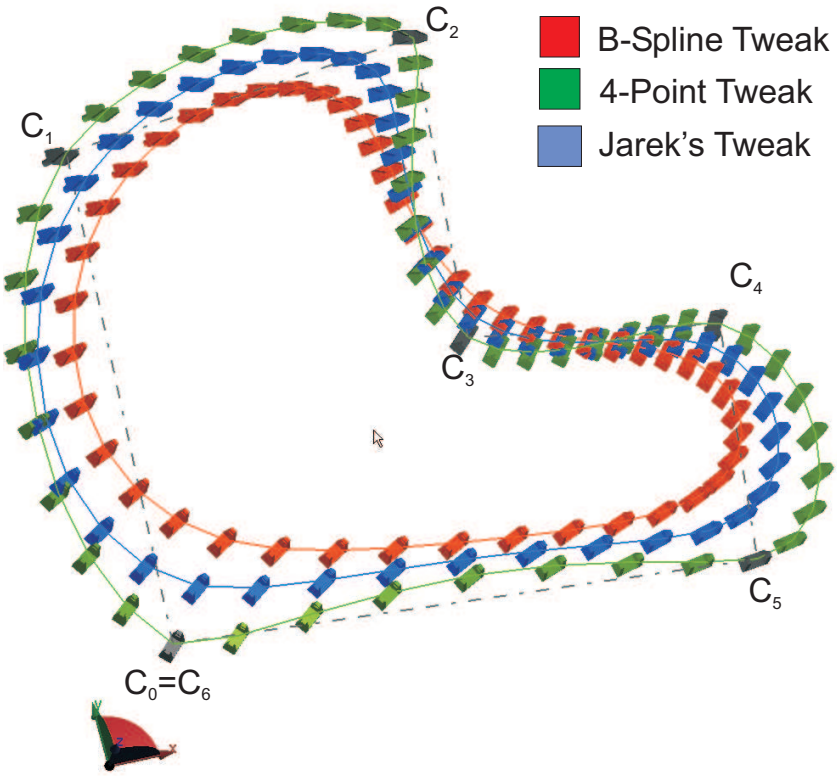


Figure 3.16: Final subdivision motions after 3 iterations for a L-shaped control structure.

C_i	q_1	q_2	q_3	q_4	q_1^0	q_2^0	q_3^0	q_4^0
C_0	0.0000	0.0000	0.3827	0.9238	0.6533	0.2706	0.9238	0.3827
C_1	0.0000	0.3827	0.0000	0.9239	0.3034	0.6955	2.039	-1.5307
C_2	0.2101	0.2536	-0.5190	0.9432	1.8614	4.5593	2.7499	-1.4891
C_3	0.1199	0.1293	0.3707	0.9188	2.2391	0.9653	1.9859	-1.3599
C_4	0.1504	0.0868	0.4924	0.8528	5.1623	-0.6059	0.9862	-1.4179
C_5	0.09414	0.09414	0.0789	0.9892	4.9859	0.1001	0.4233	-0.5173

Table 3.7: Dual quaternion representation of L-shaped polygon used for plotting various Subdivision motions

Chapter 4

Using MoDes Software

4.1 Getting started with MoDes

Install MDL package on your system and start it by double-clicking on the shortcut icon of **MoDes** on the desktop of your computer. The MDL package can be downloaded from the URL provided in Appendix B. You can also choose **Start ➠ All Programs ➠ MDL ➠ MoDes** from the taskbar menu. Once you launch the application, you will see the screen as shown in Fig. 4.1.

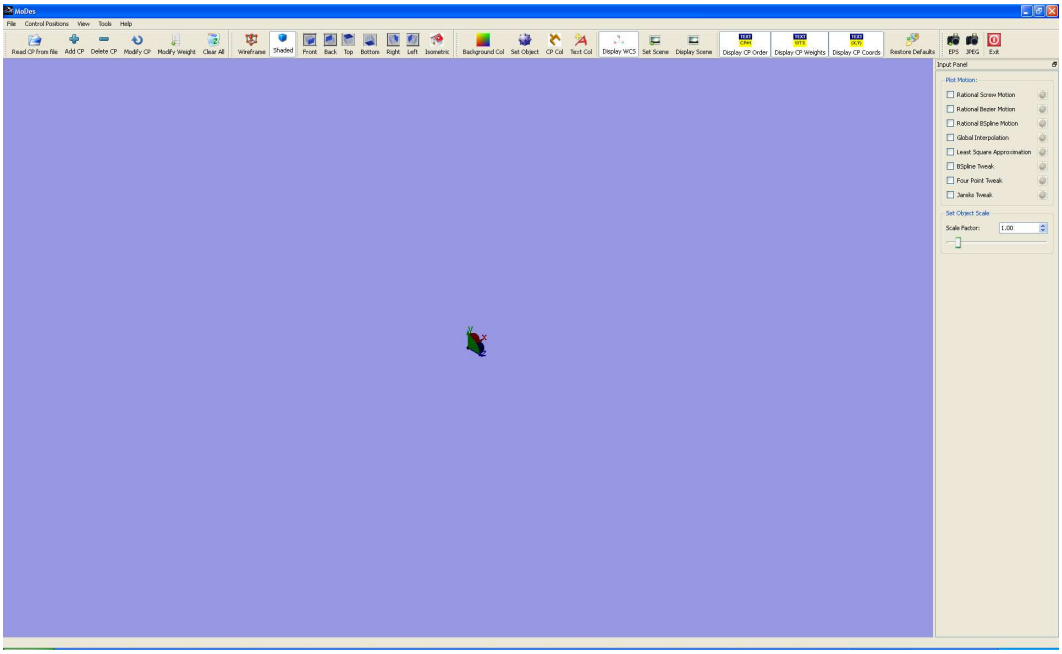


Figure 4.1: The screen that appears after launching MoDes.

It is important for the user to get acquainted with different sections of the **MoDes** Screen. The self explanatory Fig. 4.2 shows different sections with their locations.

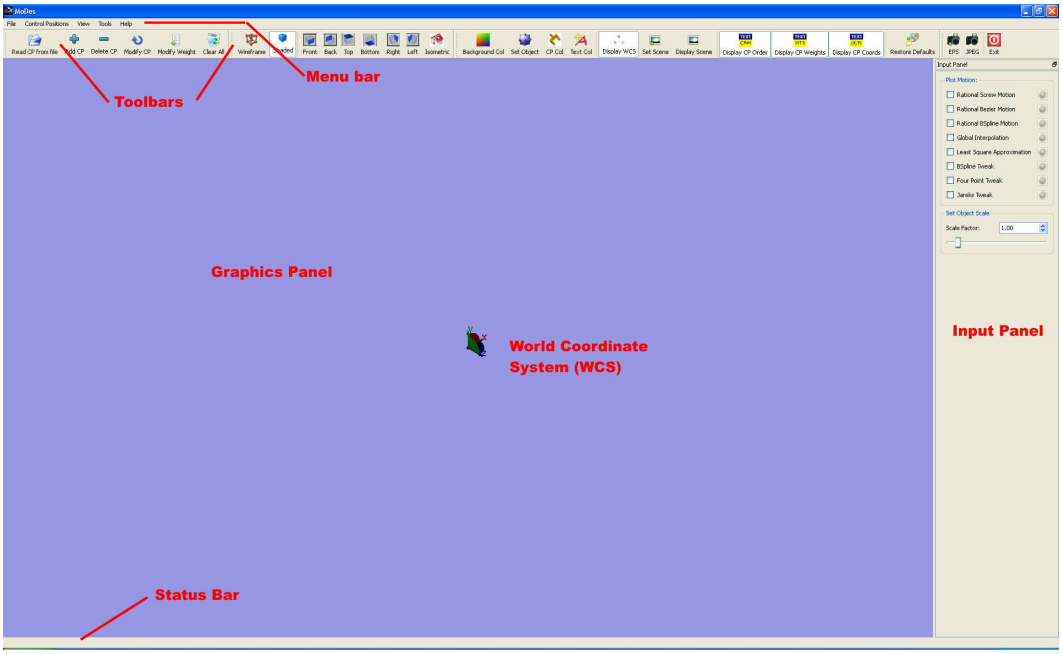


Figure 4.2: Different sections of Modes screen.

4.2 Understanding the functions of mouse buttons

As with any 3D CAD tool, it is necessary to understand the functions of mouse buttons. The efficient use of three mouse buttons, along with the CTRL and ALT key on the keyboard can can reduce the time required to complete the desired task.(see Fig. 4.3)

Figure 4.3: Functions of Mouse Buttons

- Press and hold CTRL key and then press the left mouse button in Graphics Panel to invoke the **Rotate** tool. Next drag the mouse to dynamically rotate the view.

- Press and hold CTRL key and then press the middle mouse button in Graphics Panel to invoke the **Zoom** tool. Next drag the mouse to dynamically zoom in or zoom out the view in the Graphics Panel.

- Press and hold CTRL key and then press the right mouse button in Graphics Panel to invoke the **Pan** tool. Next drag the mouse to dynamically pan the view in the Graphics Panel.

While adding Control Position's manually through user interface, combination three mouse buttons and ALT key on the keyboard can be used to set the orientation and position of the control position in 3D Space.

- Press and hold ALT key and then click and hold left mouse button to pick one of the three rotation handles on the local coordinate system

attached with the control position whose orientation and position in 3D space needs to be set. Now the drag mouse to change the orientation angle of the control position based the picked rotation handle. Depending upon on which rotation handle is selected, Control Position's angle about X, Y and Z axes of World Coordinate System can be set.(see Fig. 4.4)

- Press and hold ALT key and then press the middle mouse button. Next, drag the mouse in Graphics Panel to set the Z-Coordinate of the Control Position which is being added to the model.

- Press and hold ALT key and then press the right mouse button. Next, drag the mouse in Graphics Panel to set X and Y coordinates of the Control Position which is being added to the model.

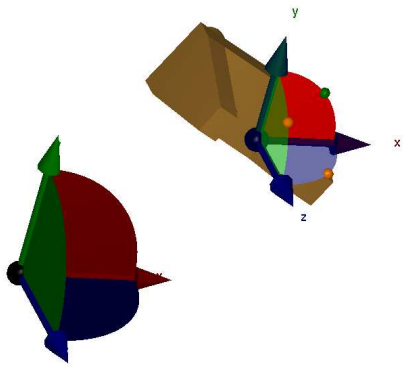


Figure 4.4: Configuring orientation of a Control Position in 3D Space: Picked rotation handle turns green. Pressing and holding ALT key and left mouse button after clicking the desired rotation handle will rotate the object about specific axes. In this case Control Position will rotate about Z-Axis.

4.3 Toolbars

MoDes offers a user friendly design environment by providing user friendly toolbars. Therefore, it is important to get acquainted with various toolbars and buttons that appear in the environment of MoDes. These toolbars are discussed next.

4.3.1 Control Positions Toolbar

This toolbar is used to carry out various operations on control positions. Fig. 4.4 shows **Control Positions** Toolbar.

Figure 4.5: Control Positions Toolbar.

It has six different tools which are described below:

1. **Read CP from file** Tool: This tool is used to read Control Positions from file. Files from which Control positions are read should have *.*data* extension. It reads the control positions from a file and and stores them in the form of dual quaternions. It can read control positions from file in three different formats MF1, MF2 and MF3. With format MF1, control positions are defined using Euler angles representing their orientation and cartesian coordinates defining the location of the origin of the moving

frame attached to the object with respect to the origin of fixed frame or world coordinate system(WCS).

Note: The file which contains control positions specified as per format MF1 must follow conventions as mentioned below:

(a) First line of the file should be MCP to indicate that file contains control positions.

(b) Second line of the file should be MF1 to indicate that the format MF1 is used.

(c) Next line should contain number of control positions to be read.

(d) From next line onwards Euler angles representing rotations about positive x, y and z axis measured in anticlockwise sense should be specified, seperated by space. Suppose 5 control positions needs to be read then next 5 lines should contain Euler angles for each control position, one set of Euler angles on each line.

(e) After that cartesian coordinates x, and z defining location of all the control positions with respect to origin should be specified, one set of cartesian coordinates on each line.

Here is a snapshot of a file which contains two control positions defined by format MF1.

```
MCP
MF1
   2

   0.00000   0.00000   45.0000
   0.00000   45.0000   0.00000



   1.00000   1.00000   0.00000
   4.00000   7.00000   0.50000
```

Figure 4.6: Snapshot of a file containing Control Positions in format MF1

With format MF2, control positions are defined using simple quaternion representing their orientation and cartesian coordinates defining their location with respect to origin.

Note: The file which contains control positions specified as per format MF2 must follow conventions as mentioned below:

(a) First line of the file should be MCP to indicate that file contains control positions.

(b) Second line of the file should be MF2 to indicate that the format MF2 is used.

(c) Next line should contain number of control positions to be read.

(d) From next line onwards simple quaternions **Q** represented by four real numbers (Q_0, Q_1, Q_2, Q_3) representing orientation of control positions should be specified. Suppose 5 control positions needs to be read then next 5 lines should contain (Q_0, Q_1, Q_2, Q_3) separated by a blank space for each control position, one set on each line.

(e) After that cartesian coordinates x, and z defining location of all the control positions with respect to origin should be specified, one set of cartesian coordinates on each line.

Here is a snapshot of a file which contains control positions defined by format MF2

```
MCP
MF2
   4

   0.68269   0.34134   0.40961   0.49964
  -0.14779   0.08867  -0.29557   0.93964
   0.00000  -0.34215   0.00000   0.93964
  -0.51085  -0.25542  -0.51085   0.64252



  -2.70000   2.00000   0.00000
  -1.00000   0.90000   1.50000
   1.50000   1.50000   1.00000
   1.50000   3.00000  -1.50000
```

Figure 4.7: Snapshot of a file containing Control Positions in format MF2

With format MF3, control positions are defined using dual quaternions.

Note: The file which contains control positions specified as per format MF32 must follow conventions as mentioned below:

(a) First line of the file should be MCP to indicate that file contains control positions.

(b) Second line of the file should be MF2 to indicate that the format MF2 is used.

(c) Next line should contain number of control positions to be read.

(d) From next line onwards simple quaternions **Q** represented by four real numbers (Q_0, Q_1, Q_2, Q_3) representing orientation of control positions should be specified. Suppose 5 control positions needs to be read then next 5 lines should contain (Q_0, Q_1, Q_2, Q_3) separated by a blank space for each control position, one set on each line.

(e) After that dual part of dual quaternions ($\hat{\mathbf{Q}}$ represented by four real numbers $(\hat{Q_0}, \hat{Q_1}, \hat{Q_2}, \hat{Q_3})$ should be specified, one set on each line. Here is a snapshot of a file which contains control positions defined by format MF3.

```
MCP
MF3
   5

   0.38268   0.00000   0.00000   0.92388
   0.00000   0.25882   0.00000   0.96593
   0.00000   0.00000   0.38268   0.92388
   0.00000   0.00000   0.00000   1.00000
  -0.38268   0.00000   0.00000   0.92388


   1.38268   0.00000   0.00000   0.92388
   0.00000   0.25882   3.00000   0.96593
   0.00000   3.00000   0.38268   0.92388
   4.00000   4.00000   4.00000   1.00000
  -1.38268   0.00000   0.00000   0.92388
```

Figure 4.8: Snapshot of a file containing Control Positions in format MF3

2. **Add CP** Tool: Invoke this tool to add Control Position to the Model manually. Once this tool is invoked a Control Position can be spotted at origin. Simultaneously a **Add/Modify Control Position** Dialogue can be seen at the bottom of **Input Panel**.

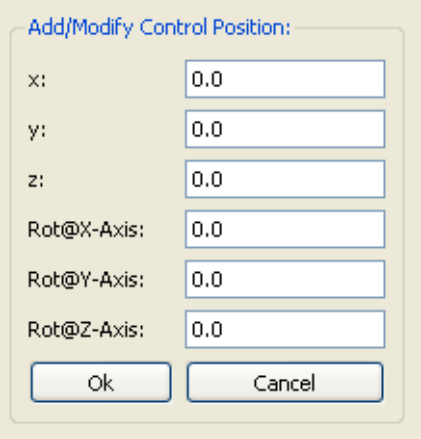


Figure 4.9: **Add/Modify Control Position** Dialogue

There are two ways in which position and orientation of Control Position which is being added can be defined. One, by specifying the parameters in the **Add/Modify Control Position** Dialogue. Second, interactively by using mouse buttons and hot keys. To define position and orientation of Control position in 3D space interactively use following key combinations:

- Press and hold ALT key and then click and hold left mouse button to pick one of the three rotation handles on the local coordinate system attached with the control position whose orientation and position in 3D space needs to be set. Now the drag mouse to change the orientation angle of the control position based the picked rotation handle. Depending upon on which rotation handle is selected, Control Position's angle about X, Y and Z axes of World Coordinate System can be set.

- Press and hold ALT key and then press the middle mouse button. Next, drag the mouse in Graphics Panel to set the Z-Coordinate of the Control Position which is being added to the model.
- Press and hold ALT key and then press the right mouse button. Next, drag the mouse in Graphics Panel to set X and Y coordinates of the Control Position which is being added to the model.

While adding the Control Position interactively, as position and orientation is changed respective fields in **Add/Modify Control Position** Dialogue are updated accordingly.

3. **Delete CP** Tool:Invoke this tool to delete Control Position. Click on the Control Position you intend to delete. The selected Control Position will be highlighted by a green bounding box as shown in Fig. 4.10

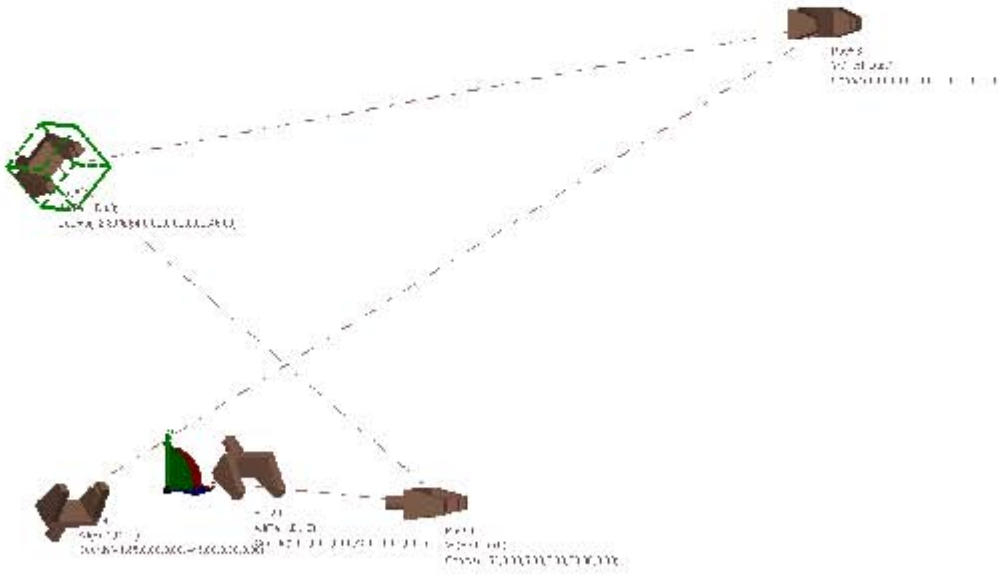


Figure 4.10: Figure showing selected control position which is highlighted by a green bounding box

Confirm **Delete CP** operation by clicking on `Ok` button of the **Delete Picked Control Position** dialogue. The dialogue appears at the bottom of **Input Panel**. Click on `Cancel` to discard the operation.

Figure 4.11: **Delete Picked Control Position** dialogue.

4. **Modify CP** Tool: this tool is used to modify the position as well as orientation of Control Positions. Invoke this tool and then Click on the

Control Position you intend to modify. The selected Control Position will be highlighted by a green bounding box as shown in Fig. 4.10. Once a Control Position is picked, **Add/Modify Control Position** dialogue will be displayed at the bottom of **Input Panel** with its fields defining current position and orientation. Modify field values to define the new position and orientation. Click on `Ok` button to apply the changes. Click on `Cancel` button to discard the operation.

5. **Modify Weight** Tool: This tool is used to modify dual weights associated with the Control Positions. It is important to note that when Control positions are read from file or when added manually their dual weights are initialized to unity i.e; real part = 1.0 and dual part = 0.0. By using this tool real and dual part of weights can be set to different values. Invoke this tool and and then Click on the Control Position whose weight you intend to modify. The selected Control Position will be highlighted by a green bounding box as shown in Fig. 4.10. Once a Control Position is picked, **Modify Weight** dialogue will be displayed at the bottom of **Input Panel** with its fields defining current real and dual values of weight associated with it. Modify field values to define the new dual weight of the picked Control Position. Click on `Ok` button to apply the changes. Click on `Cancel` button to discard the operation.

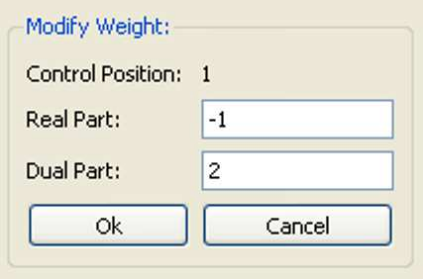


Figure 4.12: **Modify Weight** dialogue.

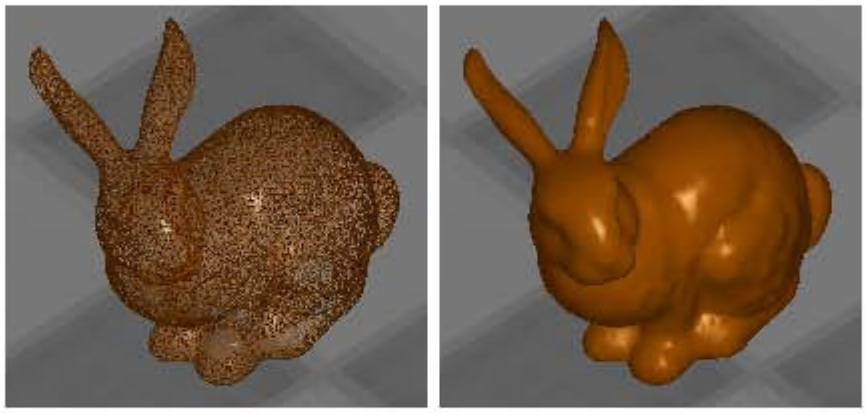
6. **Clear All** Tool: This tools clears all the Control Positions from the model.

4.3.2 View Toolbar

Figure 4.13: View Toolbar.

1. **Wireframe** Tool: This tool turns on wireframe display. If the model contains large number of control positions and intermediate positions in a given motion, then while carrying out viewing transformations such as Pan/Zoom/Rotate can cause heavy computational load because of more number of calculations involved in **Shaded** mode. So using this tool can help in smooth viewing transformation operations.See Fig. 4.14.

2. **Shaded** Tool: This tool turns on shaded display. Turning this option might result in a jerky viewing transformations if there are large number



(a) Wireframe Display (b) Shaded Display

Figure 4.14: Display modes

of Control Positions and intermediate positions in a given motion. In such case it is recommended that object with lesser number of vertices such as Cube should be used. Toggling to wireframe mode can also improve performance. See Fig. 4.14.

3. **Front** Tool: Sets eye position so as to view front side.

4. **Back** Tool: Sets eye position so as to view back side.

5. **Top** Tool: Sets eye position so as to view top side.

6. **Bottom** Tool: Sets eye position so as to view bottom side.

7. **Right** Tool: Sets eye position so as to view right side.

8. **Left** Tool: Sets eye position so as to view left side.

9. **Isometric** Tool: Sets eye position so as to get isometric view.

4.3.3 Settings Toolbar

Figure 4.15: Setting Toolbar.

1. **Background Col** Tool: this tool is used to set the background color of the **Graphics Panel**. It invokes the **Select Color** Dialogue, which can be used to pick the desired color. See Fig. 4.16.

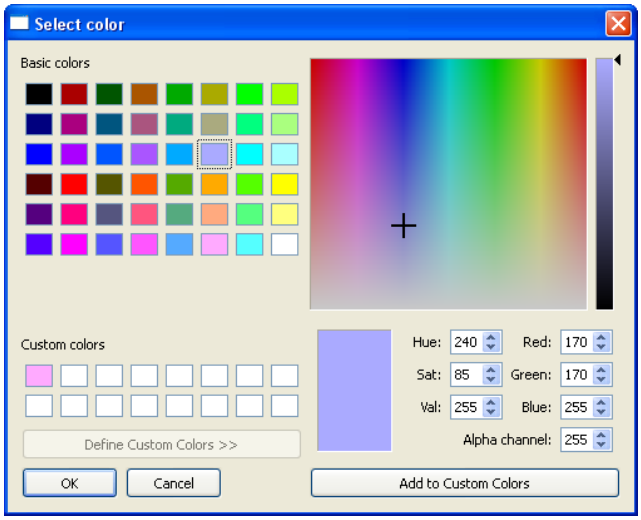


Figure 4.16: **Select Color** Dialogue.

2. **Set Object** Tool: This tool invokes **Set object for Plotting Motion** dialogue, which can be used to pick from one of the eleven different types of objects for plotting motion. See Fig. 4.17. **Robotic End effector** is the default object. User can also pick famous data set in Computer

Graphics community, **Stanford Bunny** for plotting motion. However, because of large number of vertices (about 60,000) in this model, rendering and viewing transformations can be slow.

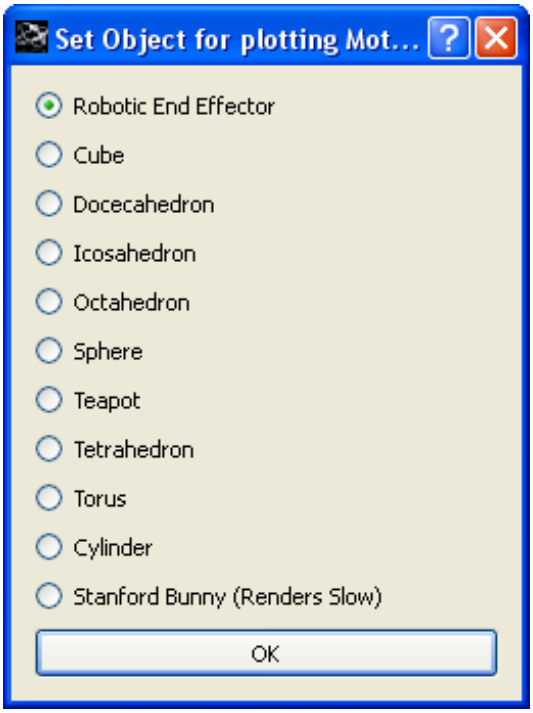


Figure 4.17: **Set Object for Plotting Motion** Dialogue.

User can set size of the objects by using **Set Object Scale** dialogue available on **Input Panel**. See Fig. 4.18.

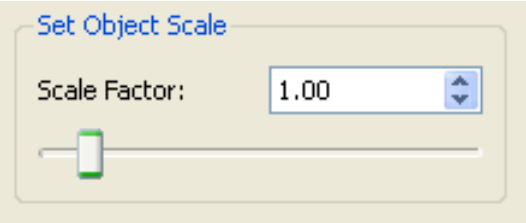


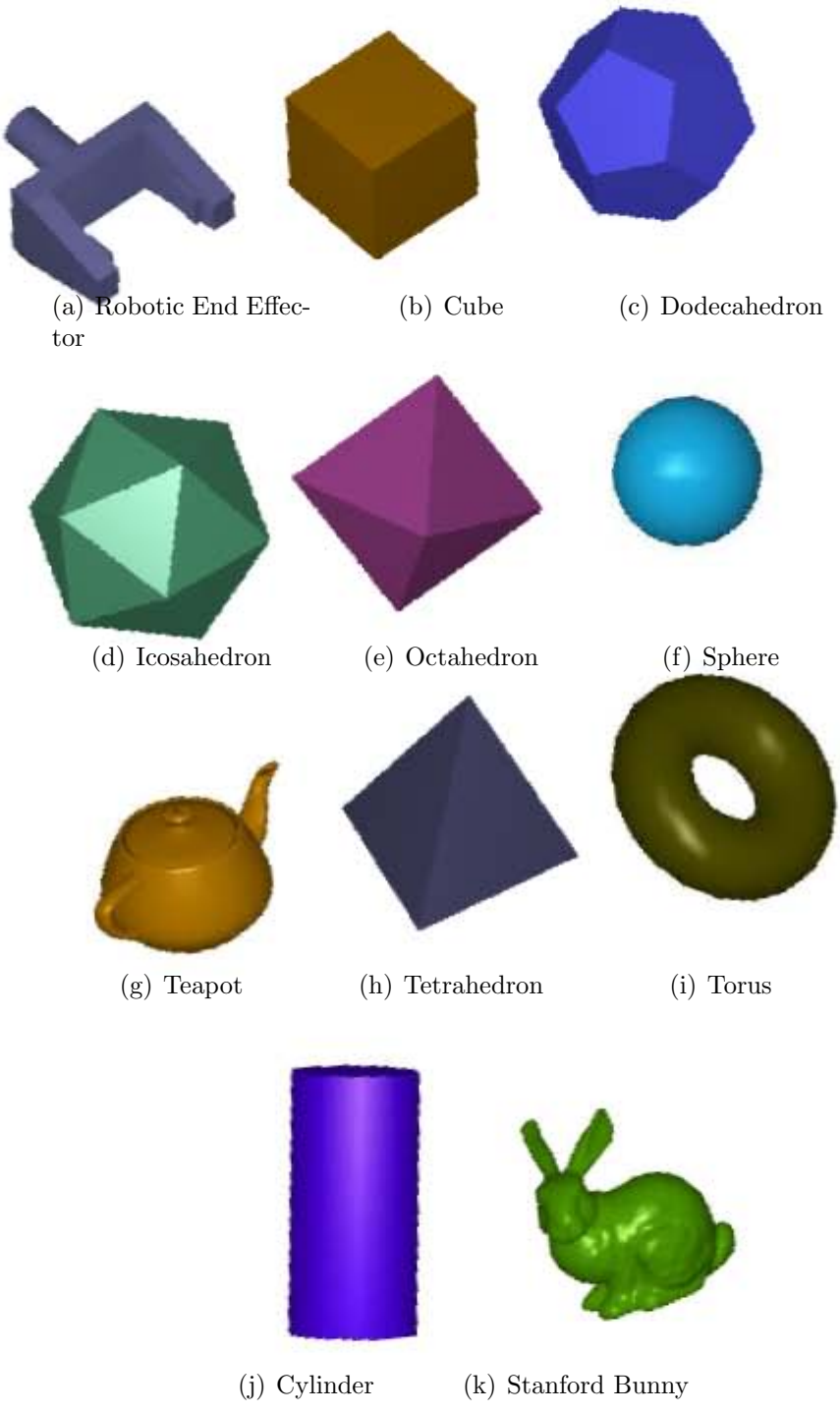
Figure 4.18: **Set Object Scale** Dialogue.

Fig. 4.19 shows different types of objects that user can pick for plotting motion. In future this list can be extended if required.

3. **CP Col** Tool: This tool is used to set the color of the Control Positions. It invokes the **Select Color** Dialogue, which can be used to pick the desired color. See Fig. 4.16.

4. **Text Col** Tool: This tool is used to set the text color. It invokes the **Select Color** Dialogue, which can be used to pick the desired color. See Fig. 4.16.

5. **Display WCS** Tool: This tool toggles on and off display of WCS i.e; World Coordinate System. See Fig. 4.20.



(a) Robotic End Effector (b) Cube (c) Dodecahedron

(d) Icosahedron (e) Octahedron (f) Sphere

(g) Teapot (h) Tetrahedron (i) Torus

(j) Cylinder (k) Stanford Bunny

Figure 4.19: Different objects available for plotting motion

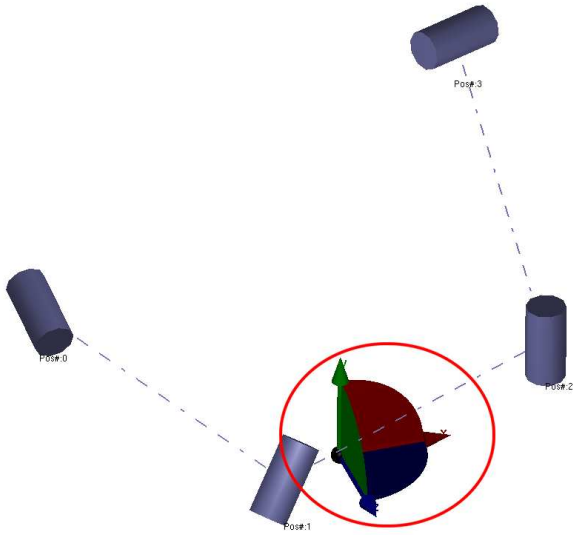


Figure 4.20: World Coordinate System (WCS)

6. **Set Scene** Tool: This tool invokes **Scene** Dialogue. See Fig. 4.21.Using this tool user can set the dimensions as well as center of the Scene so as to house all Control Positions and Motion. User can also selectively toggle display of Front Wall, Back Wall, Left Wall, Right Wall, Ceiling and Floor, for getting desired effect.

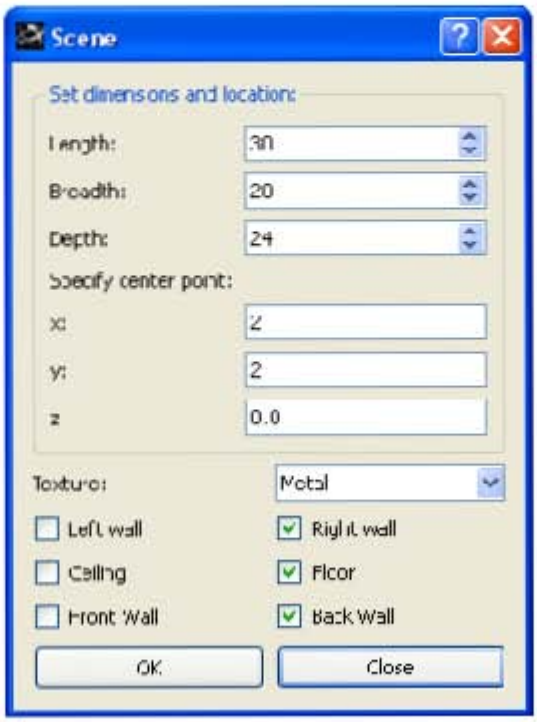


Figure 4.21: **Scene** Dialogue

There are seven different texture available at the moment for plotting a scene. In future, developers can add more textures with better resolutions if required. Fig. 4.22 shows a rational B-Spline Motion with Display of Scene toggled on with wooden texture.

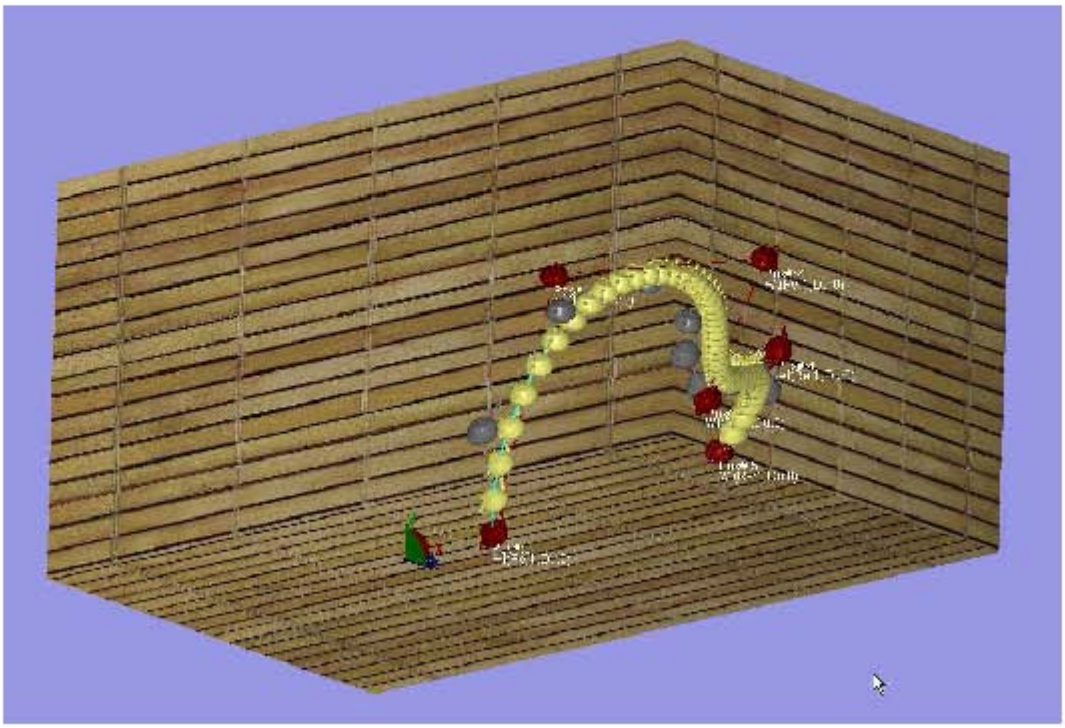


Figure 4.22: Rational B-Spline Motion with display of Scene toggled on with wooden Texture.

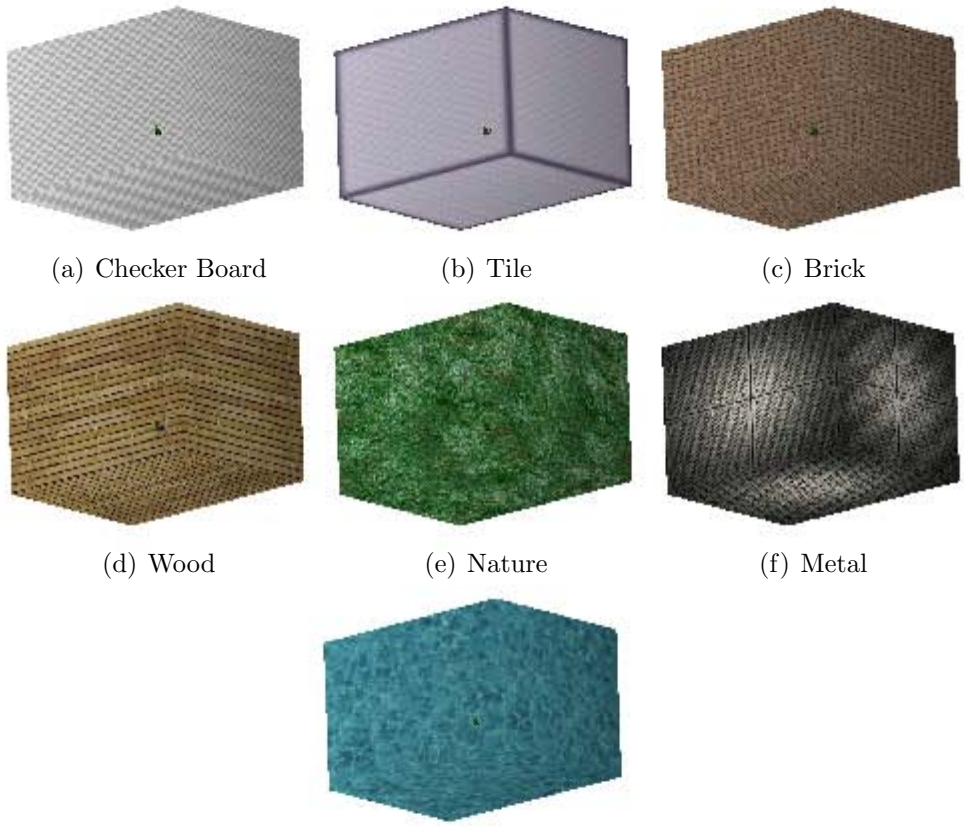
See Fig. 4.23 for view of available textures for plotting a Scene.

7. **Display Scene** Tool: This tool toggles on and off display of Scene.

8. **Display CP Order** Tool: This tool toggles on and off display of text showing order of Control Positions in **Graphics Panel**. See Fig. 4.24.

9. **Display CP Weights** Tool:This tool toggles on and off display of text showing weights of Control Positions in **Graphics Panel**. See Fig. 4.24.

10. **Display CP Coords** Tool: This tool toggles on and off display of text showing coordinates of Control Positions in **Graphics Panel**. See Fig. 4.24.

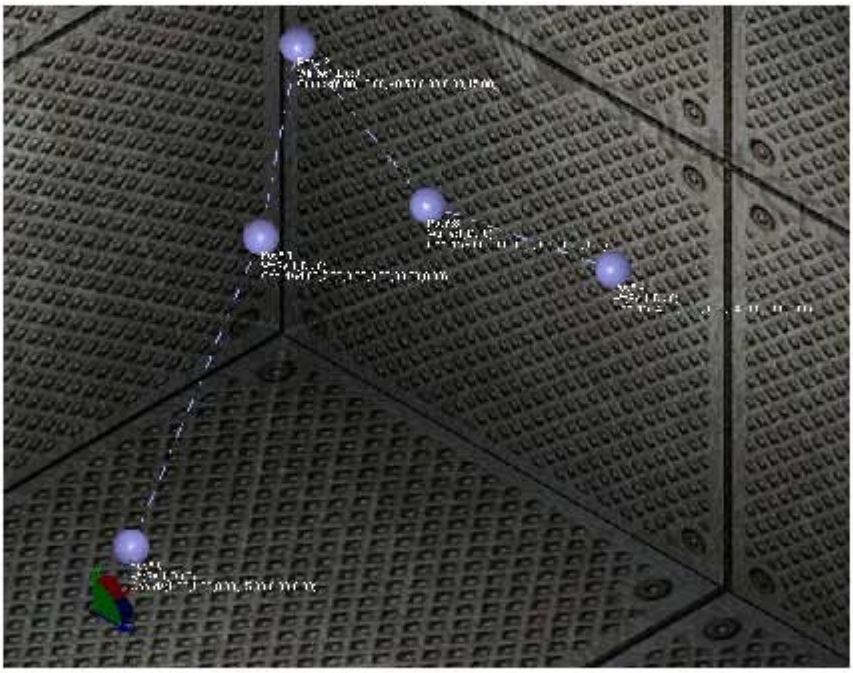


(a) Checker Board (b) Tile (c) Brick

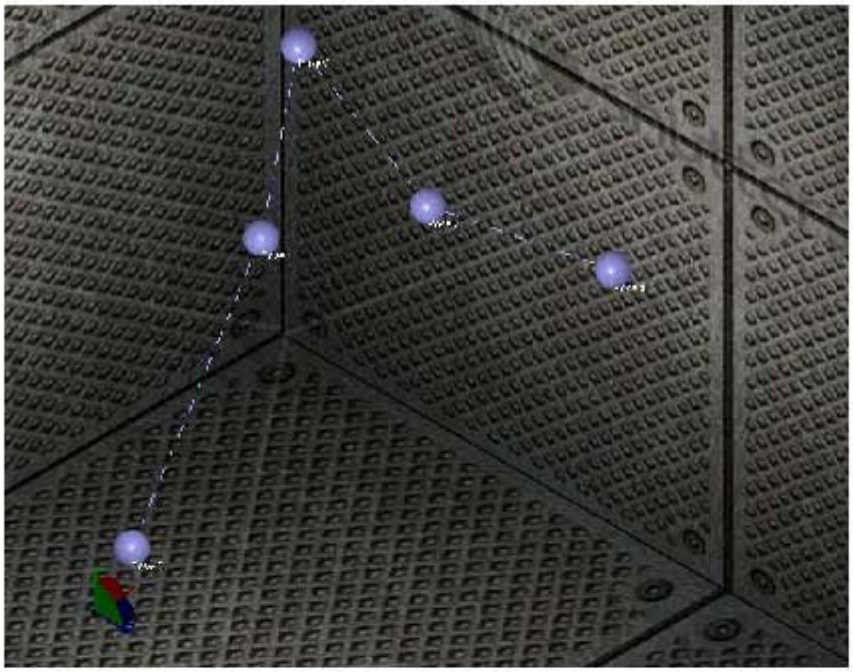
(d) Wood (e) Nature (f) Metal

(g) Water

Figure 4.23: Different textures available for plotting Scene



(a) Display of all text is turned on.



(b) Display of text showing CP Weights and CP Coords is turned off.

Figure 4.24: Toggling display of Text in **Graphics Panel**

11. **Restore Defaults** Tool: This tool is used to restore the default settings of the program.It resets the **Graphics Panel's** background Color, Text Color, Control Position Color to program default. It also resets the Object type to **Robotic End Effector**, turns on the display of text showing CP Order, CP Weights and CP Coordinates. It also turns off the display of **Scene**, if it was enabled.

4.3.4 File Toolbar

Figure 4.25: File Toolbar.

1. **EPS** Tool: This tool generates a *eps* file of the current view in **Graphics Panel**. This tool is aimed at generating high quality graphics, especially for publication purpose.

2. **JPEG** Tool: This tool generates an *jpeg* file of the current view in **Graphics Panel**. Files saved in JPEG format can be very handy for generating reports and presentation using popular tools such as Microsoft Word and Powerpoint. However, *eps* file format is recommended for generating high quality images.

3. **Exit** Tool: As name suggests, it terminates the program.

4.4 Plotting Motion

To plot a motion first read the control positions from file or add them interactively. Then check the motion which you want plot from the **Plot Motion** dialogue on the **Input Panel**. See Fig. 4.26.

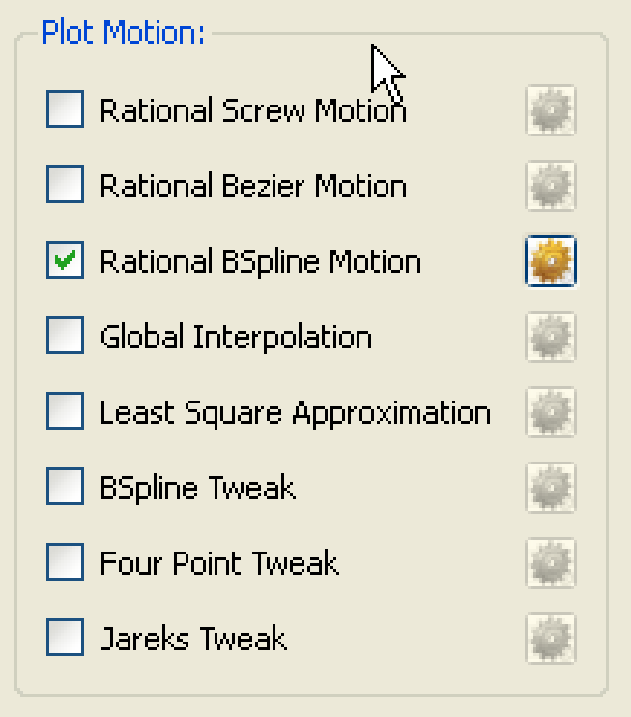


Figure 4.26: Plot Motion Dialogue

Multiple motions can be plotted simultaneously by checking multiple motions.

To set the various motion parameters click on the settings button next to the name of the motion in the **Plot Motion** dialogue. A dialogue box with allows access to various motion parameters and settings will pop up.

Fig. 4.27 shows one such dialogue box for rational B-spline motion. Modify the parameters as required. Press `Ok` to plot the motion with modified parameters. Press `Cancel` to discard the operation.

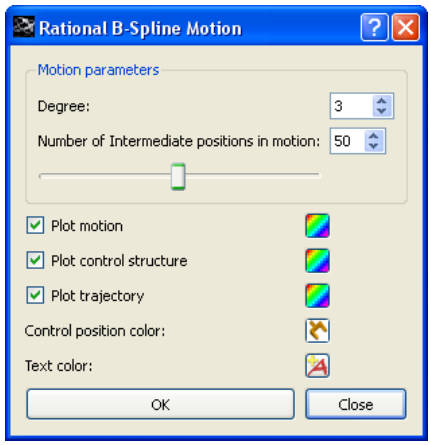


Figure 4.27: Dialogue box for setting parameters of Rational B-Spline Motion

Chapter 5

Extending Motion Design Library

One of the prime motive behind developing Motion Design Library was to release it as open source code, so that researchers, students and other professionals from CAD,Mechanical Design and Software Development Industry can add new motion design algorithms to the existing library and make it a comprehensive library which can be used in solving various motion design problems.

Following sections explain how Motion Design Library can be extended by adding new algorithms to it. There are basically two ways in which Motion Design Library can be extended.

1. By adding new algorithms independent of GUI developed using Qt.

2. By adding new algorithms with support of GUI developed in Qt.

Both approaches have their own advantages and disadvantages. One advantage of adding new motion design algorithms without using support of

GUI developed in Qt is that its relatively less complex process. However, programmer is responsible for developing his own GUI by using tool of his choice for getting user input, adding/modifying control positions interactively, panning/zooming and rotating views etc. It can prove to be a very big over head and will certainly offset the advantage of ease in adding new algorithms without using support of GUI developed in Qt. In-fact more than 8000 lines of codes in Motion Design Library has been dedicated towards developing a sophisticated and user friendly GUI.

Adding new Motion Design algorithms with Qt's support can appear to be a complex process initially. But once programmer gets used to it, the results will be rewarding.

The reason behind making a distinction between Motion Design Library independent of Qt and Motion Design Library with support of Qt is to give the developer freedom to develop a graphical user interface from other GUI toolkits like GLUI, GLUT, FLTK etc., instead of Qt if necessary.

5.1 Adding new Motion Design Algorithms independent of GUI developed using Qt

Follow this steps to add a new motion design algorithm independent of GUI developed using Qt.

Step 1: Creating a new VC++ project: Create a new VC++ project and name it as MDLv1 for instance. Add all the header files and source files found in `Mlib` and `QLib` directories which stands for Motion Library and

Quaternion Library respectively. Adding filters as like `QLib` and `MLib` helps in managing files better and speeds up development process. See Fig. 5.1. In addition add a file called `main` which contains the function `main`. Various functions to set up OpenGL environment like `reshape`, `initGL`, `draw` etc. should be added to this file. To make the compilation process easier, make sure that you include all the header files from QLib and MLib.

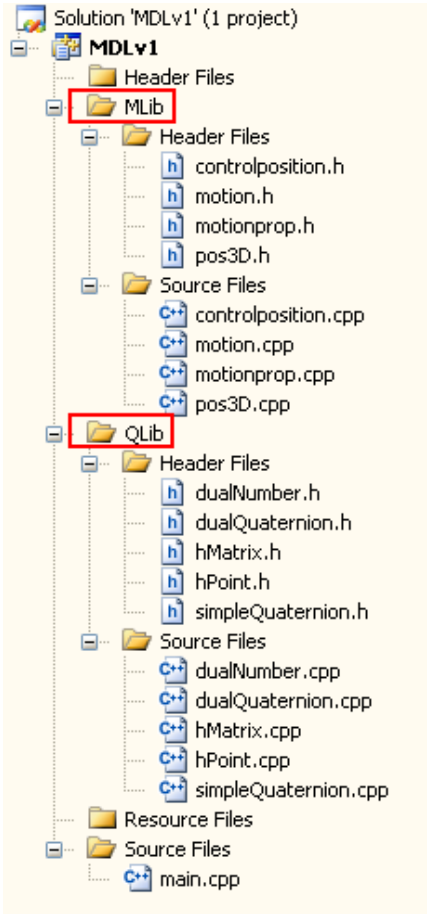


Figure 5.1: Adding proper filters while creating a new VC++ project to extend MDL can help in managing project effectively

Step 2: Declaring the class defining new motion: First Step in adding a new algorithm to Motion Design Library is to create a new class, say `MNewMotion` which inherits directly or indirectly from abstract base class `MMotion`. For adding new class you will need to modify `motion.h` and `motion.cpp` files found in `MLib` directory found in the installation directory of Motion Design Library. It is important that the programmer carefully select the class from which new class of Motion inherits. Care-

ful selection of parent class will facilitate the software reusability, which is a good software engineering practice. For example, if user wants to add a new motion design algorithm which is based principal subdivision, then it is recommended that developer inherit from `MSubDivision` class instead of abstract base class `MMotion`. By doing this developer will have access to special functions and data members specific to motions based on subdivision principal such as `splitFunc` which is used to divide the distance between two successive Control Positions in a specific ratio. Developer can also take advantage of `setIterations` and `getIterations` functions which are common to all motion design algorithms based on Subdivision principle. Following code snippet shows a pseudo code of a new class `MNewMotion` to `motion.h` file. See Fig. 5.2.

```
class MNewMotion: public MSubdivision
{
public:
     MNewMotion();     //Constructor of MNewMotion class
     ~MNewMotion();    //Destructor of MNewMotion class
     MMotionProp prop; // Instance of MMotionProp class to hold properties of motion
                       // such as motion color, various flags to decide whether
                       // motion, trajectory of motion, control structure etc.
                       // will be plotted ot not. Please refer to MMotionProp
                       // class documnetation for more details.
     //Add required public data members and functions here.
     void foo();
     virtual void plotMotion(); //Virtual function to plot motion.
                                //implement algorithm in this virtual function.
private:
    //Add required private data members and functions here
};
```

Figure 5.2: Defining a new class in `motion.h`.

In addition, add a flag of type `static bool` corresponding to this motion to `MMotion` class definition. Lets call it `NEW_MOTION`. It will be later used to decide whether a specific motion will be plotted or not.Make sure

you initialize `NEW_MOTION` to false in the beginning of `motion.cpp` file to false, as we don't want to plot any motion initially when we launch the application. Following code snippet will give you a better idea about things discussed above.

```
class MMotion
{
public:
    MMotion();
    ~MMotion();
    void setCtrlPos(vector<MDualQuat>);
    vector<MDualQuat> getCtrlPosVec();
    void setCtrlPosWtVec(vector<MDualNum> weights);
    void setWtsToUnity();
    vector<MDualNum> getCtrlPosWtVec();
    int getNoOfCtrlPos();
    void plotCtrlPos();
    virtual void plotMotion() = 0;
    double binomial(int n, int i, double t);
    double binomial(int n, int i);

    //motion
    static bool RSCREW;
    static bool RBEZIER;
    static bool RBSPLINE;
    static bool GLBL_INTERPOLATION;
    static bool LEAST_SQ_APPROX;
    static bool BSPLINE_TWEAK;
    static bool FOURPNT_TWEAK;
    static bool JAREKS_TWEAK;
    static bool NEW_MOTION;
```

Figure 5.3: Adding `NEW MOTION` flag to `MMotion` class definition

Step 3: The next step is to define the functions declared in the new class, say `MNewMotion` in `motion.cpp` file, which can be located in `MLib` directory found in the installation directory of Motion Design Library. The main purpose behind declaring the class in `motion.h` and defining its member

functions in `motion.cpp` is to separate interface from implementation. Following code snippet shows a pseudo code for `MNewMotion` class.See Fig. 5.4.

```
MNewMotion::MNewMotion()
{
    //Define constructor of MNewMotion class
    //..
    //..
}

MNewMotion::~MNewMotion()
{
    //Define destructor of MNewMotion class
    //..
    //..
}

void MNewMotion::foo()
{
    //define some helper function foo() here.
    //..
    //..
}

void MNewMotion::plotMotion()
{
    //implement algorithm to plot motion in this virtual function.
    //..
    //..
}
```

Figure 5.4: Defining member functions of new class in `motion.cpp`.

Following is a real example. `MJareksTweak` is a class which inherits from `MSubDivision` class and it is used for designing motion based on Jarek's Tweak subdivision algorithm. See Fig. 5.5 for class definition of

`MJaerksTweak` in `motion.h` file

```
class MJareksTweak: public MSubdivision
{
public:
     MJareksTweak ();
     ~MJareksTweak ();
      MMotionProp prop;
     vector<MDualQuat> jareksTweak_Algorithm(vector<MDualQuat>);
     virtual void plotMotion();
private:
};
```

Figure 5.5: `MJaerksTweak` class definition in `motion.h`.

Fig. 5.6 and Fig. 5.7 shows code that defines member functions of `MJareksTweak` class. Notice how different types of classes from Motion Design Library are used to implement the algorithm.

```
MJareksTweak::MJareksTweak()
{
    this->setIterations(3);
    float motionCol[4] = {0.67, 1.0, 0.5, 1.0};
    prop.setMotionColor(motionCol);
}

MJareksTweak::~MJareksTweak()
{
}

vector<MDualQuat> MJareksTweak::jareksTweak_Algorithm(vector<MDualQuat> control_pos)
{
    vector<MDualQuat> midP0;
    vector<MDualQuat> midP1;
    vector<MDualQuat> rearranged_midP1;
    vector<MDualQuat> tweaked1;
    vector<MDualQuat> tweaked2=control_pos;
    int sizeOfVec = control_pos.size();
    if (sizeOfVec>2)
    {
       midP0= MSubdivision::splitFunc( control_pos, 0.5);
       midP1= MSubdivision::splitFunc( midP0, 0.5);
       //rearrangement of elements in midP1
       for (int i=0; i<sizeOfVec; i++)
       {
           if(i==0)
              { rearranged_midP1.push_back(midP1[i+midP1.size()-1]);}
           else
              {rearranged_midP1.push_back(midP1[i-1]);}
       }
       //while calling this function, order in which we pass
       //vectors is important. split (v1,v2,ratio)!= split (v2,v1,ratio)
       tweaked1=MSubdivision::splitFunc(control_pos, rearranged_midP1, 0.25 );
       midP0.push_back(midP0[0]);
       for (int i=0; i<sizeOfVec; i++)
       {
          if(i==0)
          {
              tweaked2[i]=midP0[i]+0.125*(midP0[i] - 0.5*(midP0[i+midP0.size()-2]+midP0[i+1]));
          }
          else
          {
              tweaked2[i]=midP0[i]+0.125*(midP0[i] - 0.5*(midP0[i-1]+midP0[i+1]));
          }
       }
       tweaked2.push_back(tweaked2[0]);
       control_pos.clear();
       for (unsigned int i=0; i<sizeOfVec;i++)
       {
          control_pos.push_back(tweaked1[i]);
          control_pos.push_back(tweaked2[i]);
       }
       return(control_pos);
    }
}
```

Figure 5.6: `MJaerksTweak` class's member function definitions in `motion.cpp`(Part 1).

```
void MJareksTweak::plotMotion()
{
   int noOfCtrlPos = getNoOfCtrlPos();
   //process only if there are more than two control positions
   if (noOfCtrlPos>2)
   {
            vector<MDualQuat> newCtrlpos = this->getCtrlPosVec();
            vector<MDualQuat> control_pos = this->getCtrlPosVec();
            int noOfIter =  this->getIterations();
            control_pos.push_back(control_pos[0]);
            for(int k=0;k<control_pos.size()-1;k++)
             {
                  MMatrix4x4 temp1 = control_pos[k].dualQuatToTransMat();
                  MMatrix OpenGLMat = MMatrix4x4::trasformationMatToOpenGLMat(temp1);
                  float BLACK[3] = { 0.0f, 0.0f, 0.0f };
                  glMaterialfv(GL_FRONT_AND_BACK,GL_AMBIENT_AND_DIFFUSE,BLACK);
                  glPushMatrix();
                  glMultMatrixd((const GLdouble*) OpenGLMat.m[0]);
                  plotObject(); glPopMatrix();
            }
            for (int i=0; i<noOfIter; i++)
            {
                  newCtrlpos=jareksTweak_Algorithm(newCtrlpos);
            }
            vector<vector<double>> transInterPos; //store translation components of inter pos for plotting traj.
            for(int i=0;i<newCtrlpos.size()-1;i++)
            {
                  MMatrix4x4 temp = newCtrlpos[i].dualQuatToTransMat();
                  MMatrix OpenGLMat = MMatrix4x4::trasformationMatToOpenGLMat(temp);
                  if (prop.isPlotTrajectory())
                  {
                     vector<double> trans;
                     trans.push_back(OpenGLMat.m[0][12]);  //x
                     trans.push_back(OpenGLMat.m[0][13]);  //y
                     trans.push_back(OpenGLMat.m[0][14]);  //z
                     transInterPos.push_back(trans);
                  }
                  if (this->prop.isPlotMotion()==true)
                  {
                        glMaterialfv(GL_FRONT_AND_BACK,GL_AMBIENT_AND_DIFFUSE,prop.getMotionColor());
                        glPushMatrix();
                        glMultMatrixd((const GLdouble*) OpenGLMat.m[0]);
                        plotObject();  glPopMatrix();
                  }
            }
            if (prop.isPlotTrajectory())
            {
               glDisable(GL_LIGHTING);
               glColor4fv(prop.getTrajectoryColor());
               glLineWidth(2.0);
               glBegin(GL_LINES);
               for (int i=0; i< newCtrlpos.size()-2  ; i++)
               {
                     glVertex3d(transInterPos.at(i).at(0), transInterPos.at(i).at(1), transInterPos.at(i).at(2));
                     glVertex3d(transInterPos.at(i+1).at(0), transInterPos.at(i+1).at(1), transInterPos.at(i+1).at(2));
               }
               //complete the loop
               glVertex3d(transInterPos.at(newCtrlpos.size()-2).at(0), transInterPos.at(newCtrlpos.size()-2).at(1),
                                          transInterPos.at(newCtrlpos.size()-2).at(2));
               glVertex3d(transInterPos.at(0).at(0), transInterPos.at(0).at(1), transInterPos.at(0).at(2));
               glEnd();  glEnable(GL_LIGHTING);

         }
   }
}
```

Figure 5.7: `MJaerksTweak` class's member function definitions in `motion.cpp`(part 2).

Step 4: Once the class is declared and defined, next step is to create an instance of the class and plot the motion using function which is used drawing object in OpenGL. Let us name the function as `draw` and let

the name of function which performs initialization operation for OpenGL be `init`. Programmer can use `init` function to read the control positions from file. Programmer can also ask user to input Control Positions through Console Window. Its up to the programmer to decide how he wants to get Control Positions from the user. All the Control Positions needs to be saved using an instance of `MCtrlPos` class or a pointer to the `MCtrlPos` class. In this case a pointer to the `MCtrlPos`, `ctrlPosPtr` is used. Next, in the `draw` function dynamically allocate memory to a pointer `newMotion` which points to `MNewMotion` class. Use `setCtrlPos` function to assign read/entered Control Positions to `newMotion`. Use `setCtrlPosWtVec` function to set weights of Control Positions. please note that, all weights are initialized to unity unless modified. Now call the `plotmotion` function to plot the Motion. Finally free the memory allocated dynamically to the pointer `newMotion`. Failure to do so may result in large amount of memory leak as `draw` function is called repeatedly in OpenGL implementation. See Fig. 5.8

```
#include <iostream>
#include <GL/glut.h>
#include <vector>
#include "controlposition.h"
#include "motion.h"

MCtrlPos *ctrlPosPtr;

using namespace std;

//OpenGL initialization function.
void init(void)
{
   shellmessages();
   glClearColor(.0, .0, .0, .0);
   glShadeModel(GL_FLAT);
   glEnable(GL_POINT_SMOOTH);
   glEnable(GL_LINE_SMOOTH);
   glHint(GL_POINT_SMOOTH_HINT, GL_NICEST);
   glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);  // Antialias the lines

   ctrlPosPtr = new MCtrlPos;
   ctrlPosPtr->readCtrlPos("test.data"); //test.data is the file which
                                         //contains input Control Positions
}

// This is the function that needs to be changed for drawing.
void draw()
{
    glClear (GL_COLOR_BUFFER_BIT);
    glLoadIdentity();

    MNewMotion *newMotion = new MNewMotion;
    newMotion->setCtrlPos(ctrlPosPtr->getCtrlPos());
    newMotion->plotCtrlPos();
    delete newMotion;

    glutSwapBuffers();
}
//Define other functions including main()....
```

Figure 5.8: Procedure for plotting motion without using existing GUI developed using Qt

5.2 Adding new Motion Design Algorithms with support of GUI developed using Qt

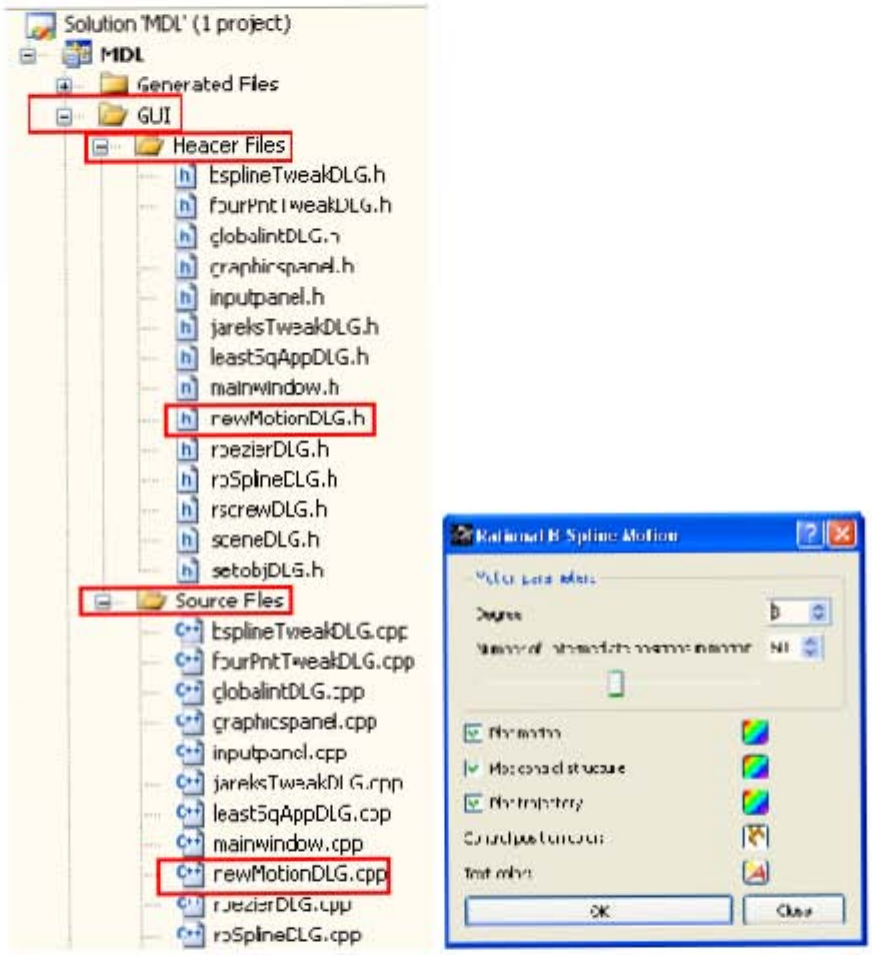
Step 1: This step is same as that for Adding new Motion Design Algorithms to the Library independent of Qt.

Step 2: This step is same as that for Adding new Motion Design Algorithms to the Library independent of Qt.

Step 3: Once the class for plotting a new motion is defined, programmer gets a better idea about expected input from the user. So programmer is in better position to design the user interface for getting input from user. To create a dialogue box for getting user input for various parameters necessary for plotting motion create two new files say `newmotionDLG.h` and `newMotion.cpp` and add them under **MDL ➠ GUI ➠ Header Files** and **MDL ➠ GUI ➠ Header Files ➠ Source Files** respectively. Refer Fig. 5.9

While designing dialogue box, define the proper Signals and Slots to define actions to be taken when certain buttons are pressed. Fig. 5.9 shows dialogue box for setting various parameters of Rational B-Spline Motion.

Step 4: Next step is to add the newly added motion to **Input Panel** so that user can selectively decide to plot a specific motion. User should also be provided with tool so that he can invoke dialogue to set various parameters for the motion. This can be done by



(a) Adding files which are used for designing Dialogues Boxes to MDL project

(b) Rational B-Spline Motion Dialogue

Figure 5.9: Adding new dialogues corresponding to new motion being added

adding checkbox and a settings button to **Plot Motion** group box on input panel. Fig. 5.10 shows how a checkbox for plotting a `New Motion` an adjacent button providing access to settings for plotting `New Motion` can be added to **Plot Motion** group box on **Input Panel**.

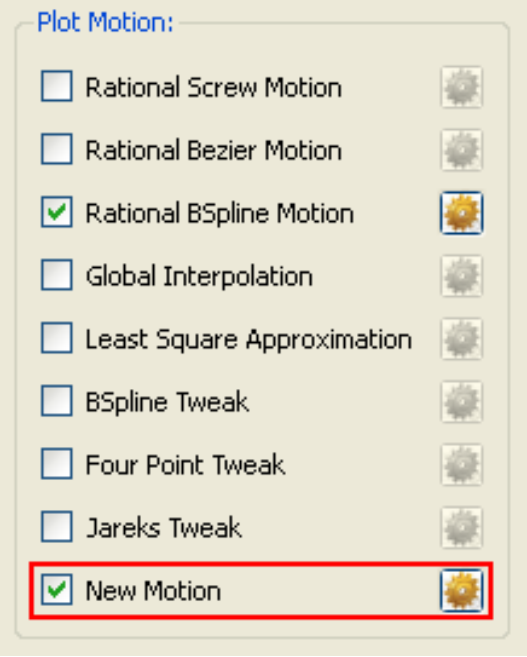


Figure 5.10: `New Motion` added to `Plot Motion` group box

To add the above mentioned check box and settings button programmer should modify `inputpanel.h` and `inputpanel.cpp` files found under `GUI` filter of VC++ project **MDL**.Open file `inputpanel.h` and add a pointer to `QCheckBox` and `QPushButton` in the public variable declaration of `InputPanel` class. In this case they are called `newMotionCB` and `newMotionBTN` respectively. See Fig. 5.11

```
//motion
QCheckBox *rscrewMotionCB;
QCheckBox *rbezierMotionCB;
QCheckBox *rbsplineMotionCB;
QCheckBox *globalIntCB;
QCheckBox *leastSqApproxCB;
QCheckBox *bsplineTweakCB;
QCheckBox *fourPntTweakCB;
QCheckBox *jareksTweakCB;
QCheckBox *newMotionCB;

QPushButton *rscrewMotionBTN;
QPushButton *rbezierMotionBTN;
QPushButton *rbsplineMotionBTN;
QPushButton *globalIntBTN;
QPushButton *leastSqApproxBTN;
QPushButton *bsplineTweakBTN;
QPushButton *fourPntTweakBTN;
QPushButton *jareksTweakBTN;
QPushButton *newMotionBTN;
```

Figure 5.11: Modifications made to `inputpanel.h` to add user interface for plotting new motion

Now modify the constructor of `InputPanel` class which can be located in `inputpanel.cpp`. Here we create a `QCheckBox` named `newMotionCB` and `QPushButton` named `newMotionBTN` and set its properties and finally add it to **Plot Motion** group box. See Fig. 5.12

```
newMotionCB =  new QCheckBox(tr("New Motion"));
newMotionBTN = new QPushButton;
newMotionBTN->setFixedSize(20,20);
newMotionBTN->setIcon(QIcon("images/motionsettings.png"));
newMotionBTN->setEnabled(true);

motionGB = new QGroupBox(tr("Plot Motion:"));
QGridLayout *motionGBLO = new QGridLayout;
motionGBLO->addWidget(rscrewMotionCB,1,0,0);
motionGBLO->addWidget(rscrewMotionBTN,1,1,0);
motionGBLO->addWidget(rbezierMotionCB,2,0,0);
motionGBLO->addWidget(rbezierMotionBTN,2,1,0);
motionGBLO->addWidget(rbsplineMotionCB,3,0,0);
motionGBLO->addWidget(rbsplineMotionBTN,3,1,0);
motionGBLO->addWidget(globalIntCB,4,0,0);
motionGBLO->addWidget(globalIntBTN,4,1,0);
motionGBLO->addWidget(leastSqApproxCB,5,0,0);
motionGBLO->addWidget(leastSqApproxBTN,5,1,0);
motionGBLO->addWidget(bsplineTweakCB,6,0,0);
motionGBLO->addWidget(bsplineTweakBTN,6,1,0);
motionGBLO->addWidget(fourPntTweakCB,7,0,0);
motionGBLO->addWidget(fourPntTweakBTN,7,1,0);
motionGBLO->addWidget(jareksTweakCB,8,0,0);
motionGBLO->addWidget(jareksTweakBTN,8,1,0);
motionGBLO->addWidget(newMotionCB,9,0,0);
motionGBLO->addWidget(newMotionBTN,9,1,0);
motionGB->setLayout(motionGBLO);
```

Figure 5.12: Modifications made to `inputpanel.cpp` to add user interface for plotting new motion

Step 5: Next Step is to create an instance of `NewMotionDLG` class created previously, in `mainwindow.h` file which can be located in **MDL ➠ GUI ➠ Header Files** filter of **MDL** project. In this file create a pointer `newMotionDLG` to `MNewMotionDLG` class and add a declare slot named `newMotionSlot`. This slot is used to decide what course of action should be taken when settings button next to `None Motion` check box is clicked. See Fig. 5.13

```
setObjDLG = 0;
sceneDLG = 0;
rscrewDLG = 0;
rbezierDLG = 0;
rbSplineDLG = 0;
globalInterpolationDLG = 0;
leastSqApproxDLG = 0;
bsplineTweakDLG = 0;
fourPntTweakDLG = 0;
jareksTweakDLG = 0;
newMotinDLG = 0;
```

Figure 5.13: Modifications made to `mainwindow.h` to add user interface for plotting new motion

Open file `mainwindow.cpp`. Include the file `newMotinDLG.h`. Next, initialize `newMotionDLG` declared as pointer to `MNewMotionDLG` class to NULL pointer or zero as shown in Fig. 5.14

```
setObjDLG = 0;
sceneDLG = 0;
rscrewDLG = 0;
rbezierDLG = 0;
rbSplineDLG = 0;
globalInterpolationDLG = 0;
leastSqApproxDLG = 0;
bsplineTweakDLG = 0;
fourPntTweakDLG = 0;
jareksTweakDLG = 0;
```

Figure 5.14: Modifications made to `mainwindow.h` to add user interface for plotting new motion

Once this is done define `newMotionSlot` on the lines similar that

in shown in Fig 5.15.

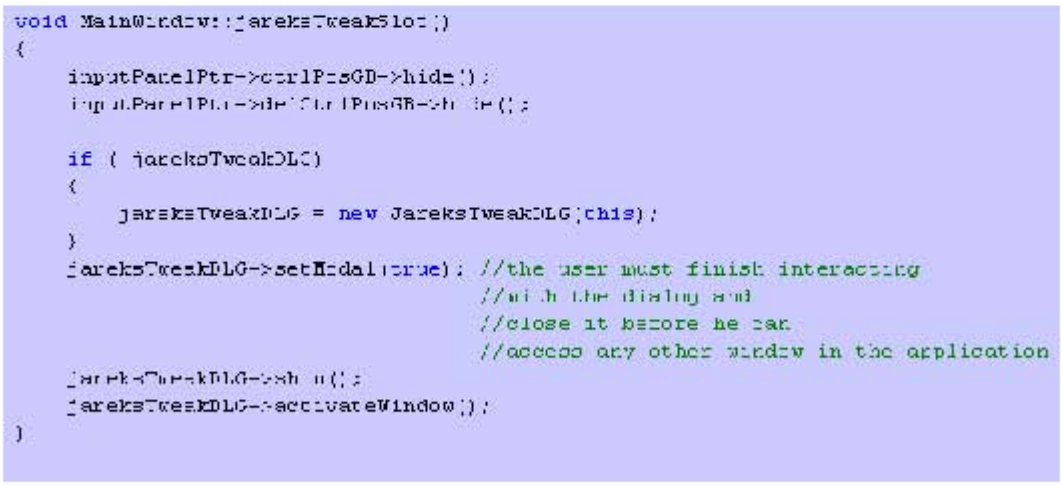


Figure 5.15: Definition of `jareksTweakSlot`

Next, locate the `createConnections` function in `mainwindow.cpp` file and make connections between `clicked` signal emitted when `newMotionCB` is clicked with the `setMotionToPlotSlot` of `GraphicsPanel` class which defines which motion will be plotted. Also connect `clicked` signal associated with `newMotionBTN` to `newMotion` slot so that `newMotionDLG` is pooped up, when user clicks on the settings button nect to `New Motion` check box in the **Plot Motion** group box.

Step 6: Last stet and most critical step to modify `graphicspanel.h` and `graphicspanel.cpp` so as to view the resulting `New Motion` in **Graphics Panel**. open file `graphicspanel.h` which can be located in **MDL ➠ GUI ➠ Header Files** filter of **MDL** project. First do

the forward declaration of class `MNewMotion` in the beginning of the file. In the declaration of the `GraphicsPanel` class add a pointer `newMotion` to the `MNewMotion` class, such that it is declared public. Now open `graphicspanel.cpp` file which can be located under **MDL ➠ GUI ➠ Source Files** filter of **MDL** project. Now in the constructor of `GraphicsPanel` class initialize `newMotion` to a NULL pointer. Now locate the `setMotionToPlot` function and add code on similar lines to that shown in Fig. 5.16. It shows the code for deciding whether to plot Jareks Tweak Motion and also avoids unnecessary memory leak.

```
if (mainWinPtr->inputPanelPtr->jareksTweakCB->isChecked())
{
    MMotion::JAREKS_TWEAK = true;
    if (jareksTweak ==NULL)
    {
       jareksTweak  = new MJareksTweak;
    }
    mainWinPtr->inputPanelPtr->jareksTweakBTN->setEnabled(true);
}
else
{
    MMotion::JAREKS_TWEAK = false;
    if (jareksTweak != NULL) //take care of memory leak
    {
       delete jareksTweak;
       jareksTweak=NULL;
    }

    mainWinPtr->inputPanelPtr->jareksTweakBTN->setEnabled(false);
}
updateGL();
```

Figure 5.16: Code snippet dealing with Jarek's Tweak Motion with in `setMotionToPlot` function

Last step is to modify the `paintGL` function found in `GraphicsPanel` class. Locate the function and add following code to plot the newly added motion. Fig. 5.17.

```
if (MMotion::JAREKS_TWEAK == true)
{
    jareksTweak->setCtrlPos(ctrlPosPtr->getCtrlPosVec());
    jareksTweak->setCtrlPosWtVec(ctrlPosPtr->getCtrlPosWtVec());
    jareksTweak->plotMotion();
}

 if (MMotion::NEW_MOTION == true)
{
    newMotion->setCtrlPos(ctrlPosPtr->getCtrlPosVec());
    newMotion->setCtrlPosWtVec(ctrlPosPtr->getCtrlPosWtVec());
    newMotion->plotMotion();
}

glFlush();
glPopMatrix();
```

Figure 5.17: Code snippet showing code that needs to be added in `paintGL` function for plotting newly added algorithm

Chapter 6

Conclusion and Future Work

This work has focussed on developing an object oriented framework for an extensible software library as well as an application which can serve as a common platform for various NURBS and dual quaternion based motion design algorithms. However, this library is not strictly restricted to NURBS class of motion and other types of motion such as those based on simple subdivision principle are also integrated into the library. This library can be broadly divided into three parts. QLib which stands for quaternion library, MLib which stands for motion library and GUI which is abbreviation for Graphical User Interface. QLib and MLib are strongly interconnected with each other. However, lot of efforts has been put to keep GUI separate from QLib and MLib. The reason behind such step was to give the developer freedom to develop a graphical user interface from free and common GUI toolkits such as GLUI, GLUT, FLTK etc., instead of Qt, if necessary.

Dual quaternion representation of spatial displacement in three dimensional space is at the heart of all the implementations of motion design schemes

in this library. Hence, a comprehensive set of functions which are dedicated to carry out various operations on dual quaternions, simple quaternions, dual numbers, homogeneous and regular matrices are developed under QLib. MLib, which stands for motion library hosts the numerous motion design algorithms. A separate class in MLib, is dedicated towards handling control positions, such a reading control position from file in three convenient formats, modifying dual weights, modifying positions interactively etc. Though developers have freedom of choosing GUI toolkit of their own choice, it is strongly insisted that developers should use and extend the current GUI developed using Qt for future developments, since a rich set of tools specific for motion design problems such as placing a control position in three dimensional space interactively, modifying weights and control positions, choosing various parameters for designing and tuning motion and many more are already carefully developed.

In my work, I have implemented eight motion design algorithms, under various classes of motions. Rational Screw Motion, Bézier motion, B-spline motion come under the fundamental motions. My implementation of above mentioned algorithms also explored the effects of dual weights and reparametrization for path invariance. Motion fitting is a critical problem in the field of motion design. Designers often seek a motion interpolating through given key frames or approximating the given set control positions. Global Interpolation to Position Data and Least Square Motion Approximation are the two algorithms which touches the problem of motion fitting. With a view that

this library can be used as educational tool by the professors and researchers working in the field of CAGD, I thought it will be good to add few motion design schemes which are simple to implement and understand. The motive behind this was to generate interest amongst the students about this subject in general. So with that point in mind, three simple subdivision based motion design schemes namely, B-spline tweak, 4-point tweak and Jarek's Tweak from a paper on Education Driven CAD by Rossignac [48] are added.

A separate chapter has been dedicated to describe how a developer can take advantage of this library to implement and test his own motion design algorithms and extend this work. In an effort to develop this library, I have written an object oriented program using VC++, Qt and OpenGL which extends more than 12,000 lines. It is a well known saying in the realm of software engineering that *If work is not documented, it does not exists.* No matter how wonderful your library is and how intelligent its design, if you're the only one who understands it, it doesn't do any good. Understanding that, a detailed documentation explaining all the classes, functions and variables extending to more than 60 pages is done. The documentation can be found in the Appendix of this thesis. Despite of all this efforts, I see my work an attempt to lay a foundation for Motion Design Library. There is a lot of scope for expansion of this library with myriads of new motion design and mechanism synthesis algorithms and fine tuning of the existing ones. A constant effort is also needed to fine polish the user interface and to add new tools which makes the designers task easier. I hope it aids researchers, students, professors as well as other

professionals working in CAGD, Computational Kinematics, Motion Design, and other related field. I also expect frequent contribution from them, small or big, so that a more comprehensive and efficient Motion Design Library can be build up on this foundation.

Bibliography

[1] Reuleaux, F., 1875. *Theoretical Kinematics: Outline of a Theory of Machines.*

[2] Bottema, O., and Roth, B., 1979. *Theoretical Kinematics.* North Holland, Amsterdam.

[3] McCarthy, J. M., 1990. *Introduction to Theoretical Kinematics.* MIT.

[4] Hunt, K., 1978. *Kinematic Geometry of Mechanisms.* Oxford University Press.

[5] Angeles, J., 1988. *Rational Kinematics.* Springer-Verlag.

[6] Farin, G., 1996. *Curves And Surfaces for Computer-Aided Geometric Design: A Practical Guide*, 4th ed. Academic Press, New York.

[7] Farin, G., 1999. *NURBS: From Projective Geometry to Practical Use.* A. K. Peters, Ltd, Natick, MA, USA.

[8] Farin, G., Hoschek, J., and Kim, M.-S., eds., 2002. *Handbook of Computer Aided Geometric Design.* Elsevier Science, North Holland.

[9] Hoschek, J., and Lasser, D., 1993. *Fundamentals of Computer Aided Geometric Design.* A K Peters.

[10] Piegl, L., and Tiller, W., 1995. *The NURBS Book.* Springer, Berlin.

[11] Gallier, J., 2000. *Curves and surfaces in geometric modeling: theory and algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[12] Farin, G., 1992. "From Conics to NURBS - a Tutorial and Survey". *IEEE Computer Graphics and Applications,* **12**(5), pp. 78–86.

[13] Piegl, L., 1986. "A geometric investigation of the rational Bezier scheme of computer-aided-design". *Computers in Industry,* **7**(5), pp. 401–410.

[14] Piegl, L., 1986. "The sphere as a rational Bezier surface". *Computer Aided Geometric Design,* **3**, pp. 45–52.

[15] Piegl, L., 1985. "Representation of quadric primitives by rational polynomials". *Computer Aided Geometric Design,* **2**, pp. 151–155.

[16] Piegl, L., 1987. "On the use of infinite control points in CAGD". *Computer Aided Geometric Design,* **4**, pp. 155–166.

[17] Piegl, L., 1987. "Infinite control points - a method for representing surfaces of revolution using boundary data". *IEEE Computer Graphics & Applications,* **7**(3), pp. 45–55.

[18] Piegl, L., 1989. "Key developments in computer-aided geometric design". *Computer-Aided Design,* **21**(5), pp. 262–274.

[19] Piegl, L., 1989. "Modifying the shape of rational B-splines .1. curves". *Computer-Aided Design,* **21**(8), pp. 509–518.

[20] Piegl, L., 1989. "Modifying the shape of rational B-splines .2. surfaces". *Computer-Aided Design,* **21**(9), pp. 538–546.

[21] Piegl, L., 1991. "On NURBS - a survey". *IEEE Computer Graphics and Applications,* **11**(1), pp. 55–71.

[22] Piegl, L., and Tiller, W., 1987. "Curve and surface constructions using rational B-splines". *Computer-Aided Design,* **19**(9), pp. 485–498.

[23] Piegl, L., and Tiller, W., 1989. "A menagerie of rational B-spline circles". *IEEE Computer Graphics & Applications,* **9**(5), pp. 48–56.

[24] Tiller, W., 1983. "Rational B-splines for curve and surface representation". *IEEE Computer Graphics & Application,* **3**(6), pp. 61–69.

[25] Böhm, W., Farin, G., and Kahmann, J., 1984. "A survey of curve and surface methods in CAGD". *Computer Aided Geometric Design,* **1**(1), pp. 1–60.

[26] Boehm, W., 1987. "Rational geometric splines". *Computer Aided Geometric Design,* **4**(1-2), pp. 67–77.

[27] Shoemake, K., 1985. "Animating rotation with quaternion curves". In Proceedings of the 12th annual conference on Computer graphics and interactive techniques, ACM Press, pp. 245–254.

[28] Pletinckx, D., 1989. "Quaternion calculus as a basic tool in computer graphics". *The Visual Computer,* **5**, pp. 2–13.

[29] Kim, M. S., and Nam, K. W., 1995. "Interpolating solid orientations with circular blending quaternion curves". *Computer-Aided Design,* **27**(5), pp. 385–398.

[30] Ge, Q. J., and Ravani, B., 1994. "Computer-aided geometric design of motion interpolants". *ASME Journal of Mechanical Design,* **116**(3), pp. 756–762.

[31] Ge, Q. J., and Ravani, B., 1993. "Computational geometry and motion approximation". In *Computational Kinematics*, J. Angeles, ed. KAP, Netherlands., pp. 229–238.

[32] Ge, Q. J., and Ravani, B., 1994. "Geometric construction of Bezier motions". *ASME Journal of Mechanical Design,* **116**(3), pp. 749–755.

[33] Park, F. C., and Ravani, B., 1995. "Bezier curves on Riemannian-manifolds and lie-groups with kinematics applications". *ASME Journal of Mechanical Design,* **117**(1), pp. 36–40.

[34] Zefran, M., Kumar, V., and Croke, C. B., 1998. "On the generation of smooth three-dimensional rigid body motions". *IEEE Transactions on Robotics and Automation,* **14**(4), pp. 576–589.

[35] Barr, A. H., Currin, B., Gabriel, S., and Hughes, J. F., 1992. "Smooth interpolation of orientations with angular velocity constraints using quaternions". *Computer Graphics,* **26**(2), pp. 313–320.

[36] Wang, W., and Joe, B., 1993. "Orientation Interpolation in Quaternion Space Using Spherical Biarcs". In Graphics Interface '93, Canadian Information Processing Society, pp. 24–32.

[37] Nielson, G. M., and Heiland, R. W., 1992. "Animated rotations using quaternion and splines on a 4D sphere". *Programming Comput. Software,* **18**, pp. 145–154.

[38] Ge, Q. J., and Ravani, B., 1993. "Motion interpolation and mechanism synthesis". In Proceedings of 1993 ASME Design Automation Conference, ASME.

[39] Ge, Q. J., and Kang, D., 1995. "Geometric design of smooth composite ruled surface strips using dual spherical geometry". In 1995 ASME Design Automation Conference ASME DE-Vol. 82:659-664.

[40] Ravani, B., and Roth, B., 1984. "Mappings of spatial kinematics". *Journal of Mechanisms Transmissions and Automation in Design-Transactions of the ASME,* **106**(3), pp. 341–347.

[41] Juttler, B., 1994. "Visualization of moving-objects using dual quaternion curves". *Computers & Graphics,* **18**(3), pp. 315–326.

[42] Jüttler, B., 1995. "Spatial rational motions and their application in Computer Aided Geometric Design". In *Mathematical Methods for Curves and Surfaces*, M. Dhlen, T. Lyche, and L. L. Schumaker, eds. Vanderbilt University Press, Nashville, pp. 271–280.

[43] Juttler, B., and Wagner, M. G., 1996. "Computer-aided design with spatial rational B-spline motions". *ASME Journal of Mechanical Design,* **118**(2), pp. 193–201.

[44] Wagner, M. G., 1994. "A Geometric Approach to Motion Design". Ph.d. dissertation, Technische Universitt Wien.

[45] Wagner, M. G., 1995. "Planar rational B-spline motions". *Computer-Aided Design,* **27**(2), pp. 129–137.

[46] Purwar, A., and Ge, Q. J., 2005. "On the effect of dual weights in computer aided design of rational motions". *ASME Journal of Mechanical Design,* **127**(5), pp. 967–972.

[47] Juttler, B., and Wagner, M., 2002. "Kinematics and Animation". In *Handbook of Computer Aided Geometric Design*, G. Farin, J. Hoschek, and M. Kim, eds. Elsevier, New York, pp. 723–748.

[48] Rossignac, J. "Education-Driven Research in CAD". *Computer Aided Design,* **36**, pp. 1461–1469.

[49] "SyMech". *http://www.symech.com/,* **Software**.

[50] "WATT". *http://www.heron-technologies.com,* **Software**.

[51] Kaufman, R., 1978. "Mechanism Design by Computer". *Machine Design,* **October**, pp. 94–100.

[52] Waldron, K. J., and Song, S. M., 1981. "Theoretical and Numerical Improvements to an Interactive Linkage Design Program, RECSYN". *Proc. of the Seventh Applied Mechanisms Conference, Kansas City, MO,* **Dec: 8.18.8**.

[53] Erdman, A., and Gustafson, J., 1977. "LINCAGES: Linkage INteractive Computer Analysis and Graphically Enhanced Synthesis Packages". *American Society of Mechanical Engineers. See also: http:// www.me.umn.edu/divisions/design/lincages/.*

[54] Larochelle P., Dooley J., M. A., and J.M., M., 1993. "SPHINX: Software for Synthesizing Spherical 4R Mechanisms". *Proceedings of the 1993 NSF Design and Manufacturing Systems Conference, University of North Carolina at Charlotte,* **Jan.**, pp. 607–611.

[55] Ruth, D. A., and McCarthy, J. M., 1997. "SphinxPC: An Implementation of Four Position Synthesis for Planar and Spherical 4R Linkages". *CD-ROM Proc. of the ASME DETC97, paper no. DETC97/DAC-3860,* **Sept.**, pp. 14–17.

[56] Furlong T. J., V. J. M., and M., L. P., 1998. "Spherical Mechanism Synthesis in Virtual Reality". *CD-ROM Proc. of the ASME DETC98, Paper No. DETC98/DAC-5584, Atlanta, GA,* **Sept.**, pp. 13–16.

[57] Furlong T. J., V. J. M., and M., L. P., 1998. "Spades: Software for Synthesizing Spatial 4C Linkages". *CD-ROM Proc. of the ASME DETC98, Paper No. DETC98/Mech- 588w9, Atlanta, GA,* **Sept.**, pp. 13–16.

[58] Alba Perez, Hai-Jun Su, J. M. M., 2004. "Synthetica 2.0: Software for the synthesis of constrained serial chains". *Proceedings of DETC04 2004 ASME Design Engineering Technical Conferences , Salt Lake City, Utah, USA,* **Sept.**

[59] Fillmore, J. P., 1984. "A note on rotation matrices". *IEEE Computer Graphics & Application,* **4**(2), pp. 30–33.

[60] Röschel, O., 1998. "Rational motion design - a survey". *Computer-Aided Design,* **30**(3), pp. 169–178.

[61] Hamilton, W. R., 1899. *Elements of Quaternions*, Vol. 1 -2. Longmans, Green & Co.

[62] Hamilton, W. R., 1853. *Lectures on Quaternions.* Hodges Smith & Co., Dublin.

[63] Waerden, B. L. v. d., 1976. "Hamilton's discovery of quaternions". *Mathematics Magazine,* **49**(5), pp. 227–234.

[64] Arunachalam, P. V., 1990. "W R Hamilton and his quaternions". *Math. Ed.,* **6**(4), pp. 261–266.

[65] Cheng, H., and Gupta, K. C., 1989. "An historical note on finite rotations". *ASME Journal of Applied Mechanics,* **56**, pp. 139–145.

[66] Pervin, E., and Webb, J. A., 1983. "Quaternions for computer vision and robotics". In International Conference on Computer Vision and Pattern Recognition, pp. 382–383.

[67] Eberly, D., 2002. Rotation representations and performance issue. Technical report, Magic Software, Inc.

[68] Dam, E. B., Koch, M., and Lillholm, M., 1998. Quaternions, interpolation and animation. Technical Report DIKU-TR-98/5, University of Copenhagen.

[69] Leandra, V., 2001. Quaternions and rotations in 3-space: the algebra and its geometric interpretation. Technical Report TR01-014, UNC Chapel Hill, North Carolina.

[70] Study, E., 1903. *Die Geometrie der Dynamen.* Verlag Teubner, Leipzig.

[71] Klein, F., and Blaschke, W., 1926. *Vorlesungen ber hhere Geometrie*, 3rd ed. Springer, Berlin.

[72] Stachel, H., 1997. "Coordinates - A survey on higher geometry". *Computer Networks and ISDN Systems,* **29**(14), pp. 1645–1654.

[73] Rath, W., 1993. "Matrix groups and kinematics in projective spaces". *Abh. Math. Sem. Univ. Hamburg,* **63**, p. 177196.

[74] Rath, W., 1996. "A kinematic mapping for projective and affine motions and some applications". In *Geometry and Topology of Submanifolds*, F. D. e. al., ed., Vol. 8. World Scientific, p. 292391.

[75] Ge, Q. J., and Sirchia, M., 1999. "Computer aided geometric design of two-parameter freeform motions". *ASME Journal of Mechanical Design,* **121**(4), pp. 502–506.

Appendix A

Installing and Compiling Motion Design Library

There are two ways in which Motion Design Library can be used. One as a stand-alone application, named as MoDes where researches, students and other professionals working in this field can generate different types of motion using various motion design algorithms already implemented in the library. The stand-alone application, MoDes can be used to view results for different sets of control positions with flexibility to set various related parameters for motion design. It can also be used to generate JPEG and EPS files of the generated motion which can be used for creating reports. Second, it can be build as a VC++ project, which can be used to add new motion design algorithms to the existing code by using the instructions provided in previous chapters.

A.1 System Requirements

The following are the system requirements to ensure the smooth running of Motion Design Library on your system

- System Unit: An Intel Pentium III or Pentium IV based system running Microsoft XP or Microsoft Vista.
- Memory: 256 MB of RAM is minimum recommended. 1 GB of RAM is recommended in general.
- Software: Microsoft Visual Studio, preferably 2005 or 2008 and Qt (Version 4.3.1 is recommended) is required for modifying and compiling the Motion Design Library. They are not required for running Motion Design Library as a stand-alone application.
- Graphics card: A graphics card with a 3D OpenGL accelerator is required, preferably with a resolution greater than 1024x768 pixels.

A.2 Installing Motion Design Library as a stand-alone application

Unzip the `MDL.rar` file to desired location. It can be found at the URL specified in the Appendix B. Double click on **MDL.exe** in the directory where `MDL.rar` is unzipped. Follow the instructions to install the Motion Design Library as a stand-alone application. After installation a shortcut will be generated on Desktop and in the Program's Menu.

A.3 Compiling Motion Design Library as a VC++ project

Motion Design Library is developed and tested using Microsoft Visual Studio 2005 and Trolltech's Qt (Version 4.3.1). Below are the step by step instructions for compiling the VC++ project **MDL**. The guidelines to build the project are given for Windows XP operating system with assumption that Microsoft Visual Studio 2005 and Qt is already installed on the user's system. Equivalent steps can be used with other operating systems such as Windows Vista and other versions of Visual Studio and Qt. However it is not yet tested on other platforms. Here are the guidelines to compile the Motion Design Library.

Step 1: Unzip the `MDL.rar` file to desired location. It can be found at the URL specified in the Appendix B. Open the VC++ project **MDL** in the directory where `MDL.rar` is unzipped.

Step 2: Before we compile the project, it is necessary that Visual Studio is integrated properly with Qt. Launch Visual Studio 2005. Go to **Tools ➠ Options ➠ Projects and Solutions ➠ VC++ Directories**.

1. Add path of `bin` directory of Qt's installation under `Executable Files` section. It can be some thing like this `C:\Qt\4.3.1\bin`.(see Fig. A.1)

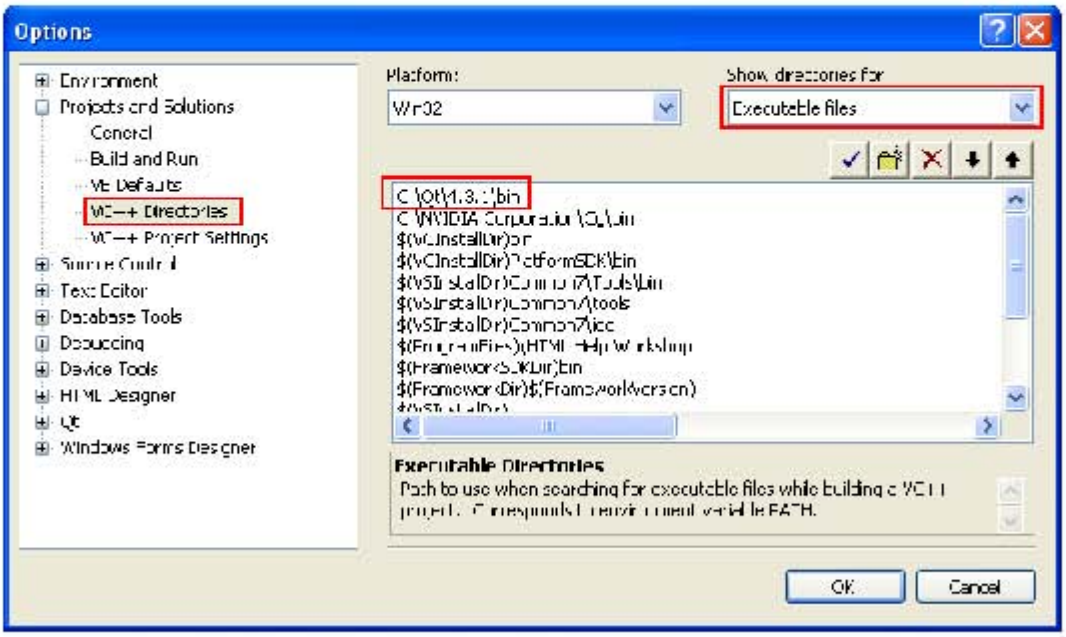


Figure A.1: Adding Qt's executable files to Visual Studio's Environment.

2. Set path of essential directories where Qt's header files are installed under `Include Files` section. Path of following three directories should be added (see Fig. A.2)

   (a) `(Qt's Installation Directory)\include`.

   (b) `(Qt's Installation Directory)\include\QtGui`.

   (c) `(Qt's Installation Directory)\include\QtOpenGL`.

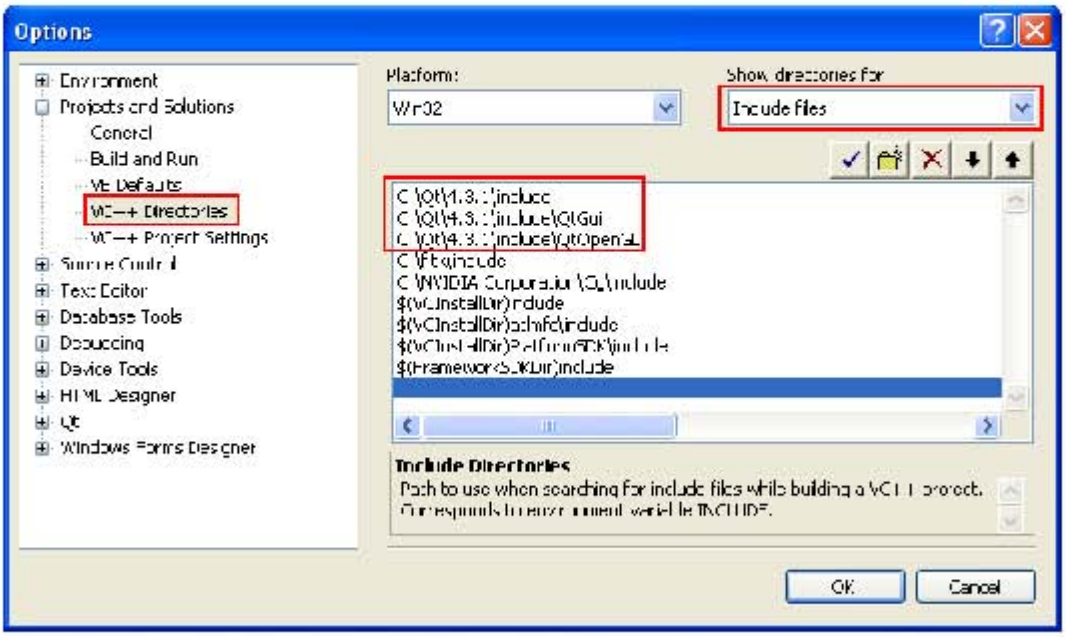


Figure A.2: Adding Qt's header files to Visual Studio's Environment.

3. Add path of `lib` directory of Qt's installation under `Library Files` section. It can be some thing like this `C:\Qt\4.3.1\lib`.(see Fig. A.3)

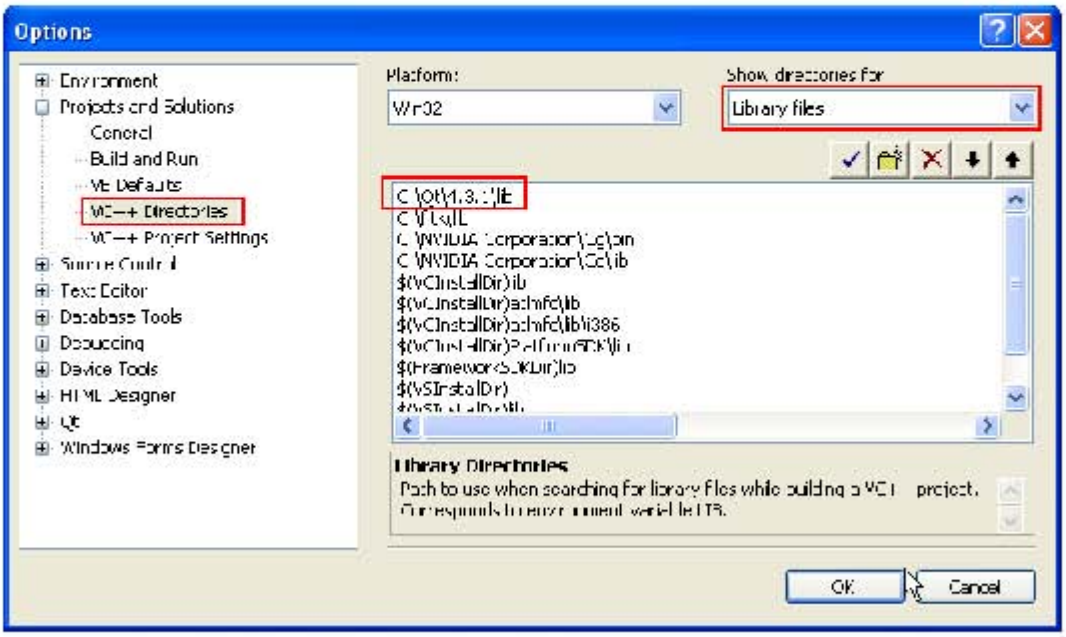


Figure A.3: Adding Qt's library files to Visual Studio's Environment.

Step 3: It is also important that the project is configured properly. To ensure that the project is configured properly, open VC++ project **MDL**. Then go to **Project** ➠ **Properties** ➠ **Linker** ➠ **Input**. In Visual Studio projects can be build in two modes namely Debug and Release. If you want to compile the project in Debug mode for testing purpose you will need to add Debug version of dependencies. In the **Additional Dependencies** field check if following library files are added. If not, add them.

- `opengl32.lib`
- `glu32.lib`
- `gdi32.lib`
- `user32.lib`

- qtmaind.lib
- QtOpenGLd4.lib
- QtGuid4.lib
- QtCored4.lib
- gl2ps.lib
- mkOpenGLJPEGImage.lib

However if you want to build the project in Release mode you will need to add Release version of dependencies. In that case **Additional Dependencies** field check if following library files are added. If not, add them.

- opengl32.lib
- glu32.lib
- gdi32.lib
- user32.lib
- qtmain.lib
- QtOpenGL4.lib
- QtGui4.lib
- QtCore4.lib
- gl2ps.lib
- mkOpenGLJPEGImage.lib

For both modes make sure that under **Ignore Specific Library** field LIBCMTD.lib is added. (see Fig. A.4)

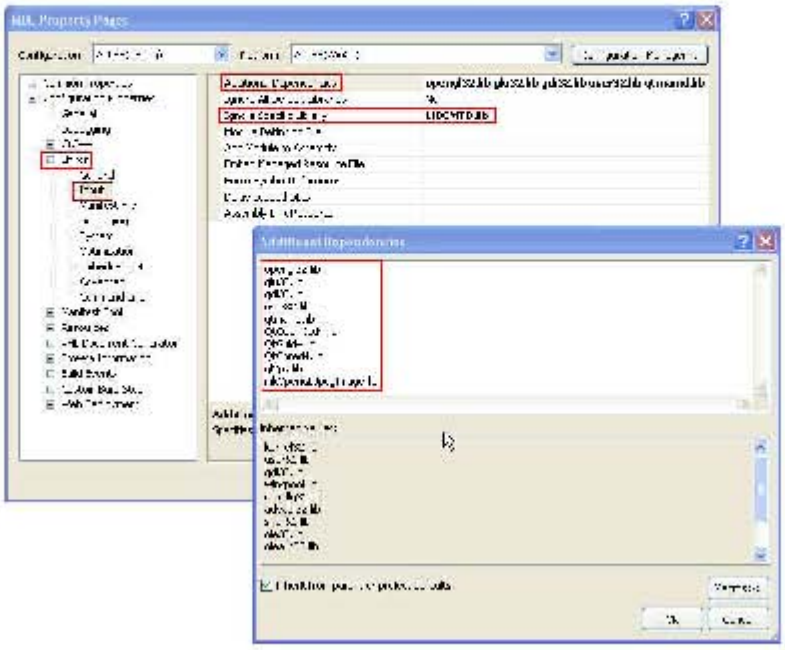


Figure A.4: Setting project properties.

That's it. Motion Design Library is now ready to be build with Visual Studio. Go to **Build ➠ Build Solution**. If every thing is configured properly the project should compile without any errors. If it does not compile, please recheck your settings.

Appendix B

Downloading MDL and Documentation

The Motion Design Library (MDL), MoDes, and Documentation can be downloaded from following URL:

macmotion.eng.sunysb.edu/MDL