

Automated Real-time Animation

By
Ting Qin

SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF COMPUTER SCIENCE

AT

LETTERKENNY INSTITUTE OF TECHNOLOGY

DONEGAL, IRELAND

NOVEMBER 2008



LETTERKENNY INSTITUTE OF TECHNOLOGY

DEPARTMENT OF

COMPUTER SCIENCE

The undersigned hereby certify that they have read and recommend to the Faculty of Science for acceptance a thesis entitled “Automated Real-time Animation” by Ting Qin in partial fulfilment of the requirement for the degree of Master of Computer Science.

Date of Submission: December 2008

Research Supervisor: Dr. Mark Leeney

Readers:

lyit | Institiúid Teicneolaíochta Leitir Ceanaínn
Letterkenny Institute of Technology

LETTERKENNY INSTITUTE OF TECHNOLOGY

Date: **December 2008**

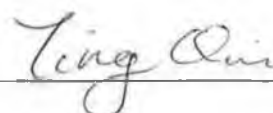
Author: Ting Qin

Title: Automated Real-time Animation

Department: Computer Science

Degree: M.Sc. Convocation: January Year: 2008

Permission is herewith granted to Letterkenny Institute of Technology to circulate and to have copied for non-commercial purposes, at its discretion, the above title upon the request of individuals or institutions.



Signature of Author

THE AUTHOR RESERVES OTHER PUBLICATION RIGHTS, AND NEITHER THE THESIS NOR EXTENSIVE EXTRACTS FROM IT MY BE PRINTED OR OTHERWISE REPRODUCED WITHOUT THE AUTHOR'S WRITTEN PERMISSION.

THE AUTHOR ATTESTS THAT PERMISSION HAS BEEN OBTAINED FOR THE USE OF ANY COPYRIGHTED MATERIAL APPEARING IN THIS THESIS (OTHER THAN BRIEF EXCERPTS REQUIRING ONLY PROPER ACKNOWLEDGEMENT IN SCHOLARLY WRITING) AND THAT ALL SUCH USE IS CLERLY ACKNOWLEDGED.

Abstract

Producing animation for the computer game/entertainment industry is a time-consuming, difficult process. One approach to maximizing the utility of previously animated sequences is to blend and warp pre-existing sequences to produce new animations on the fly. The extent to which these approaches are feasible in realtime and are doable without any animator input is debatable.

This research considers many of the approaches to blending and warping in the literature. Many of these require manual intervention to tune the transition parameters and are also computationally expensive. However, they serve as a foundation for developing an approach to the problem that is completely automatic and is computationally inexpensive.

Predefined motions of walking, running, and jumping derived from motion capture data are considered and a novel approach to producing segues between these motions is implemented.

Following the lead of research suggesting that transition times are of vital importance in the response of users to transition animations the transitions produced by the implementation were tested on a group of survey participants and the same general conclusion was supported.

To conclude the work, suggestions for improvement of the algorithms for transition production are made and possible future developments are considered.

Acknowledgments

It is a pleasure to thank the many people who made this thesis possible.

It is difficult to overstate my gratitude to my supervisor, Dr. Mark Leeney. With his inspiration, and his great efforts he helped to make animation fun for me. Throughout my thesis-writing period, he provided encouragement, sound advice, good teaching, and good company. I would have been lost without him.

I wish to thank Letterkenny Institute of Technology and Enterprise Ireland for their provision of funding and facilities, and also thank Carnegie Mellon University (CMU) Graphics Lab Motion Capture Database for providing free motion data, without which this work would not have been possible.

I wish to thank the third year class in Computer Game Development in Letterkenny Institute of Technology for their participation of the assessment and the great comments on the research artefacts.

I am grateful to the head of department Thomas Dowling and the secretaries in the Computing Department of Letterkenny Institute of Technology, especially Marnie Grier, for assisting me in many different ways.

I wish to thank my parents. They bore me, raised me, supported me, taught me, and loved me. To them I dedicate this thesis.

Abbreviations

NUBRS Non Uniform Rational Basis Spline

IK Inverse Kinematics

LERP Linear Interpolation

SLERP Spherical Linear Interpolation

Table of Contents

1. Introduction.....	1
1.1 Introduction.....	1
1.2 Objective and Questions	1
1.3 Overview of chapters.....	3
2. Literature Review	5
2.1 Introduction.....	5
2.2 Animation	5
2.2.1 Animation Techniques.....	7
2.2.1.1 Traditional animation.....	7
2.2.1.2 Stop-motion	7
2.2.1.3 Computer - Generated Animation	9
2.2.2 Modelling.....	11
2.2.2.1 Geometric Modelling.....	11
2.2.2.2 Hierarchical Model	15
2.2.3 Motion Description Methods.....	18
2.2.3.1 Keyframing.....	18
2.2.3.2 Procedural Animation.....	19
2.2.3.3 Motion Capture.....	19
2.3 Mathematics.....	23
2.3.1 Euler Angles	23
2.3.2 Quaternion	26
2.3.2.1 Angle between Two Quaternions	27
2.3.2.2 Advantages of Quaternion	28
2.3.2.3 Conversion between Euler Angle and Quaternion	28
2.3.2.4 LERP and SLERP.....	30
2.3.3 Inverse Kinematics	32
2.3.4 Catmull-Rom Splines	35
2.4 Related Works	39
2.4.1 Motion Blending and Warping.....	39
2.4.2 Motion Transition	42
2.5 Conclusion.....	43
3. Research Methodology	45
3.1 Introduction.....	45
3.2 General Research Method.....	45
3.2.1 Survey.....	47

3.3	Data Collection Methods	48
3.3.1	Observation.....	48
3.3.2	Interviews	49
3.3.3	Questionnaire.....	49
3.4	Conclusion	50
4.	Design.....	51
4.1	Introduction.....	51
4.2	Real-time Animation	51
4.3	Visualisation Tool.....	52
4.3.1	Data Format	54
4.3.2	Open Source and Drawbacks.....	55
4.4	Motion Initialization	55
4.4.1	Introduction.....	55
4.5	Blending and Warping.....	56
4.5.1	Blending.....	56
4.5.2	Warping	58
4.5.3	Position Issue.....	59
4.6	Motion Transition Duration.....	61
4.6.1	Introduction.....	61
4.6.2	Transition Duration Method	62
5.	Implementation	64
5.1	Pre-processed data	64
5.2	Blending and Warping.....	66
5.3	Catmull-Rom Application	68
6.	Evaluation.....	72
6.1	Survey Objective	72
6.2	Questionnaire Design.....	72
6.3	Results and Analysis.....	73
6.4	Comments and Issues to be Concerned	76
6.5	Conclusion.....	77
7.	Conclusions and Further Work.....	78
7.1	Conclusion.....	78
7.2	Future work.....	80
	References.....	82
	Appendix I - Motion Blending Length Survey Table.....	85
	Appendix II - Code	86

List of Figures and Tables

FIGURE 2-1 5200 YEAR OLD IRANIAN EARTHENWARE BOWL	6
FIGURE2-2 CLAY ANIMATION MOVIE ‘WALLACE AND GROMIT’	8
FIGURE 2-3 A CYLINDER IN POLYGON	12
FIGURE 2-4 A SPLINE (NURBS) HAND MODEL BY ERIC MASLOWSKI 2005 ERIC@EGO- FARMS.COM	13
FIGURE 2-5 NURBS CONTROL POINT AND CONTROL POLYGON.....	15
FIGURE 2-6 SKELETON IN FRONT VIEW	16
FIGURE 2-7 TREE STRUCTURE OF SKELETON	17
FIGURE 2-8 MOTION CAPTURE SYSTEM BY HTTP://VRLAB.EPFL.CH/RESEARCH/LO_LOCOMOTIN_ENGINE.HTML	21
FIGURE2-9 EULER ANGLE FROM WORLFRAM MATHWORLD	24
FIGURE 2-10 LINEAR INTERPOLATION	30
FIGURE2-11 SPHERICAL LINEAR INTERPOLATION	31
FIGURE 2-12 INVERSE KINEMATICS MECHANISM BY STEWART DICKSON IN “DIGITAL CHARACTER CONSTRUCTION”	33
FIGURE 2-13 FOOT-SLIDE.....	34
FIGURE 2-14 CUBIC CURVES	35
FIGURE 2-15 CATMULL-ROM SPLINE	36
FIGURE 4-1 3D VIEWER INTERFACE	53
FIGURE 4-2 TRANSITION SEGMENTS.....	57
FIGURE 4-3 KEY FRAME SETUPS	57
FIGURE 4-4 ROOT POSITION CALCULATION	61
TABLE 2-1PRICE COMPARISON OF MOTION CAPTURE SYSTEM ON THE MARKET	22
TABLE 4-1 SURVEY RESULT OF TRANSITION PERIOD	74

1. Introduction

1.1 Introduction

Animation is costly and time-consuming. One approach to creating animation is to utilise motion captured data by concatenating two library motion sequences. Motion captured data is realistic and can be used when developing computer games and movies. However, it has some drawbacks and inconveniences; the actor may not produce the desired motion, the data may be noisy or it may not directly apply to the avatar due to the differences in body dimensions or other parameters. Furthermore, motion capture systems are relatively costly. In an attempt to overcome these difficulties, animators began manipulation of existing data by use of blends and warps. However, these methods appear to be time-consuming and require a significant amount of expert user input. Recent studies,(Kovar 2003; Sang 2004; Qiang 2006) suggest that the manual input could be eliminated by use of automatic methods. However, the complete solution for automated animation is not yet available.

1.2 Objective and Questions

Investigation of ways to automate blends and warps in real-time could help animators in seamless blending of two motion clips without manual inputs and modifications. Outcomes of this research could contribute to the area of real-time animation, particularly in interactive environments, such as computer games.

The purpose of this research is to investigate ways of fully or partially automating computations in order to produce smooth, realistic-looking transitions between motion clips without manual input.

Main Research Question(s):

Can automated blending of similar motions, such as walk and run be achieved at interactive rates, thereby increasing the utility of a database of motion without user input?

Can current work on blending/warping be extended to deal with a wider range of motion types?

Can artefacts of blending/warping such as foot skate or self-intersection be eliminated at interactive speed?

Objectives:

- Develop automated approaches to performing blending of pre-existing motion sequences.
 - Produce automated approaches to warping a pre-existing motion sequence to fit a set of warp constraints.
 - Incorporate steps in blending and warping procedures to eliminate unwanted artefacts.
-

- Derive an automatic scheme for deciding if two sequences can be successfully blended under a given time constraint and/or whether a natural-looking motion warp can be achieved within a given time constraint.
- Analyse the applicability of above blending methods when applied to dissimilar sequences.

1.3 Overview of chapters

Chapter Two

This chapter aims to identify background information in the area of animation. It begins with introduction of animation, animation techniques, character modelling, and motion description methods. Next, the research explores some background mathematics, such as Euler angles, quaternions and inverse kinematics. Finally, the study investigates some other related researchers' work in the area of motion blending and warping and transition period.

Chapter Three

The purpose of this chapter is to provide a comprehensive description of the research methodology used for this study. The chapter provides an outline of major research philosophies, potential strategies and data collection instruments. It also justifies why the use of a survey is the appropriate data collection instrument for this particular section. The limitations of the research are also outlined.

Chapter Four

Chapter four outlines the development process of this research. It introduces advantages and disadvantages of the open-source visualisation tool and data format of the file used in this research. Then, it discusses pre-processing of data in order to obtain a desired data set, which can perform the transition. It is followed by blending and warping settings which divide transition duration into six sections and have key frames in each section. It then focuses on how the outcomes of the survey are used to obtain blending length. Catmull-Rom splines are then used to smooth the motion.

Chapter Five

The purpose of this chapter is to present the key conclusions reached as a result of the conducted research. The chapter discusses the implications of the findings and highlights potential areas for further research.

2. Literature Review

2.1 Introduction

This chapter focuses on the background information of the area of animation. It begins with introduction of animation, animation techniques, character modelling, and motion description methods. Next, the research explores some background mathematics. Finally, it investigates other related researchers' work in the area of motion blending and length of transition.

2.2 Animation

“To animate is, literally, to bring to life.”(Foley 1997) Animation covers all changes that have a visual effect. It thus includes the time-varying position, shape, colour, structure and texture of an object, and changes in lighting, camera position, orientation and focus, and rendering techniques.(Foley 1997) The major uses of animation are in the entertainment industry. Animation is also a form of art in film production. It is often displayed in film festivals throughout the world. In addition, there has been a growing use of instructional and educational animation such as control systems and flight simulators for aircraft, and in scientific research. In animation industry, the following companies are known as the pioneers.

- Pixar (Toy Story (1995), Toy Story 2 (1999), Finding Nemo (2003));
- Dreamworks (Shrek(2001), Shrek 2 (2004))
- Disney (Pirates of the Caribbean: The Curse of the Black Pearl (2003), The Incredibles (2004))
- Square-Enix (Final Fantasy X, XI (2001,2002) (PS2))
- Eidos Interactive (Lara Croft Tomb Raider: The Angel of Darkness, Thief: Deadly Shadows)

The first concepts of animation can be found when studying small devices and objects crafted and used centuries ago. Devices such as thaumatropes, zoetropes or flip books were used to present motion in the early stages of animation. The 5200 year old Iranian earthenware bowl is the world's oldest example of animation (see Figure 2-1)(Ball 2008). When the bowl is spun, it shows a goat leaping up to a tree to take a pear.



Figure 2-1 5200 year old Iranian earthenware bowl

The now traditional form of animation was developed in the early 1900s and refined by Ub Iwerks, Walt Disney and others. One second of animation requires up to 24 distinct drawings. The majority of animations come from professional

animation studios because they are very time-consuming and expensive to produce.

2.2.1 Animation Techniques

2.2.1.1 Traditional animation

Traditional animation, also called cell animation, is the process that was used for development of the majority of animated films of 20th century. Di Fiore describes traditional animation as *“the process of creating a sequence of drawn images which, when shown one after the other at a fixed rate, resembles a lifelike movement.”*(Di Fiore 2001) The individual frames of a traditionally animated film are produced by taking photographs of drawings. Examples are Fantasia (1940, USA, Disney), Toy Story (1995, USA, Disney/Pixar), The Triplets of Belleville (2003, France, Sony Pictures Classics).

2.2.1.2 Stop-motion

Stop-motion animation is the technique that creates one frame at a time by physically manipulating and photographing real-world objects. There are many different types of stop-motion animation, usually named after the type of media used to create the objects, such as clay animation, puppet animation and go motion.

Clay animation uses figures made of clay or a similar malleable material to create stop-motion animation. The figures are similar to the puppet animation that can be manipulated in order to pose the figures.



Figure2-2 Clay animation movie 'Wallace and Gromit'

Puppet animation generally uses an armature inside the puppet to position and move particular joints. An example is the TV series Robot Chicken (US, 2005).

Go motion uses various techniques to create motion blur between frames of film, which is not present in traditional stop-motion. The technique was invented by

Industrial Light and Magic and Phil Tippett to create special effects scenes for the film *The Empire Strikes Back* (1980).

2.2.1.3 Computer - Generated Animation

Computers can be used to produce new applications and tools for animation to give animators more capabilities. They use either traditional or new methods to produce animation. New animation techniques include skeletal animation, per-vertex animation, cel-shaded animation, onion skinning, analogue computer animation, and motion capture. Computer-generated animation can be divided into two categories: two dimensional and three dimensional. Two-dimensional animation techniques tend to focus on image manipulation while three-dimensional techniques usually build virtual worlds in which characters and objects move and interact.

Two-dimensional animation techniques contribute to computer animation by providing the tools used for sprite-based animation, blending or morphing between images, embedding graphical objects in video footage, or creating abstract patterns from mathematical equations.

The most common form of two-dimensional animation is sprite animation. A sprite is a bitmap image or set of images that are composited over a background, producing the illusion of motion. Sprite-based animation can be done extremely quickly with current graphics hardware, and thus many elements of the scene can

move simultaneously. The disadvantage of this technique is that changes in lighting and depth cannot be reproduced. Thus, sprite animation is most often used in interactive media where rendering speed is more important than realism.

Computer animation in three dimensions often involves constructing a virtual world where characters may move and interact. Human characters in such a virtual world are described as virtual humans and their movement and interaction as behaviours, see animation Shrek (2001 DreamWorks).

Many applications require realistic, high-quality character animation. Applications as diverse as simulation, movies, and video games depend on natural-looking motion. Applications commonly demand sequences that transition between types of motion (such as skipping to running) or between particular frames in a motion collection. There may be several constraints on the transition, but typically the transition must look realistic and be of a particular duration.

The motions of a virtual character are usually created by motion capture techniques. The captured motions are reproductions of real human motions and therefore the most realistic. However, the captured motions also lack flexibility in situations where the human model or the environment is different from the time when the motions were captured. Therefore, making captured motions more reusable for different characters and environments has become a focus of recent work.

2.2.2 Modelling

2.2.2.1 Geometric Modelling

There are many examples of models in physical and social sciences. According to (Foley 1997), types of models for which computer graphics are used are organizational models, quantitative models and geographic models. Geographic models describe components with natural geometrical properties. A geometric model may represent:(Foley 1997)

- Layout and attributes affecting the appearance of components, such as shape.
- Structure or topology which can be specified in a matrix for networks or in a tree structure for a hierarchy, or may have its own intrinsic geometry.
- Specific data values and properties associate with components, such as descriptive text.

The models describe the process of forming the shape of an object. The two most common sources of 3D models are those originated on the computer by an artist or engineer using certain 3D modelling tools, and those scanned into a computer from real-world objects. Models can also be produced procedurally or via physical simulation. 3D modelling needs to be performed by a dedicated program or an application component. The most popular and influential programs are Maya, 3DS Max, Blender, Lightwave, Modo and others. Different modelling methods as described below are used to present or approximate a model by using the above programs:

Polygon - A vertex is a point in 3D space. Two vertices form a straight line called an edge. Three vertices, connected to the each other by three edges, form a polygon. However, polygons cannot be bent. Curved surfaces are approximated by using many small flat surfaces. The vast majority of 3D models today are built as textured polygonal models, because they are the most flexible and quickest for the computer to handle.

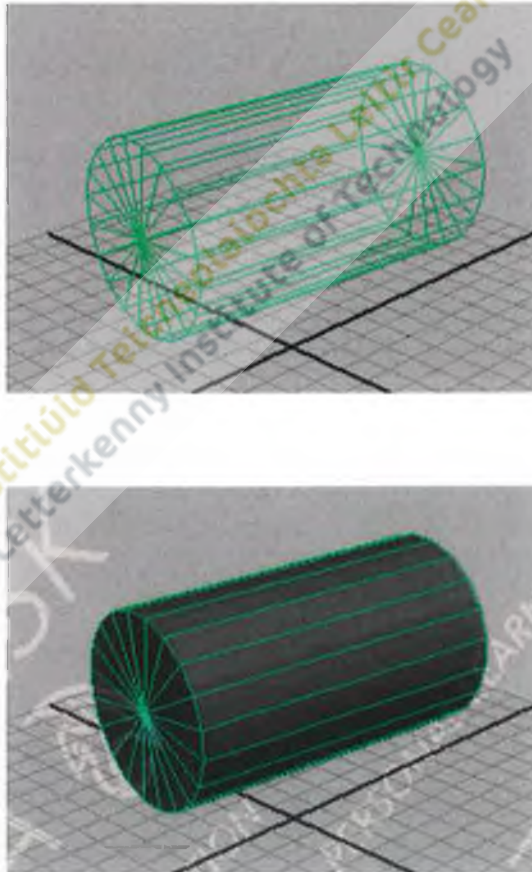


Figure 2-3 A cylinder in polygon

Spline – Maya defines a spline as “In general, a curved line, made up of segments and defined by control points. Types of splines include polylines, cardinal splines,

B-splines, and non-uniform rational B-splines (NURBS).” Splines tend to produce smoother results than polygons. Splines are well suited to creating complex shapes such as human faces, weapons, and spacecraft. Splines are often better for applications like this because their method of building forms uses smooth and natural curves, rather than uneven and artificial polygonal shapes.



Figure 2-4 A spline (NURBS) hand model by Eric Maslowski 2005 Eric@ego-farms.com

Non Uniform Rational Basis Spline (NURBS) is one of most popular used splines. NURBS surfaces are truly smooth surfaces, not approximations using small flat surfaces like polygon modelling. NURBS have been the standard for virtually all high-end modelling work due to their implicit UV texture space, resolution independence and intuitive curve-derivation. (Les 1997) defines NURBS as following. A p th-degree NURBS curve is defined by:

$$C(u) = \frac{\sum_{i=0}^n N_{i,p}(u) w_i P_i}{\sum_{i=0}^n N_{i,p}(u) w_i} \quad a \leq u \leq b \quad \text{[Equation 2-1]}$$

where the $\{P_i\}$ are the control points (forming a control polygon), the $\{w_i\}$ are the weights, and the $\{N_{i,p}(u)\}$ are the p th-degree B-spline basis function defined on the nonperiodic (and nonuniform) knot vector

$$U = \left\{ \underbrace{a, \dots, a}_{p+1}, u_{p+1}, \dots, u_{m-p-1}, \underbrace{b, \dots, b}_{p+1} \right\} \quad \text{[Equation 2-2]}$$

Unless otherwise stated, we assume $a=0$, $b=1$, and $w_i > 0$ for all i .

Setting

$$R_{i,p}(u) = \frac{N_{i,p}(u) w_i}{\sum_{j=0}^n N_{j,p}(u) w_j} \quad \text{[Equation 2-3]}$$

Allows us to rewrite Equation 2-1 as

$$C(u) = \sum_{i=0}^n R_{i,p}(u) P_i \quad \text{[Equation 2-4]}$$

The $\{R_{i,p}(u)\}$ is the rational basis functions; they are piecewise rational functions on $u \in [0, 1]$.

NURBS curves and surfaces are generalizations of both B-splines and Bézier curves and surfaces, the primary difference being the weighting of the control points which makes NURBS curves *rational* (non-rational B-splines are a special case of rational B-splines). Figure 2-5 illustrates the structure of NURBS.

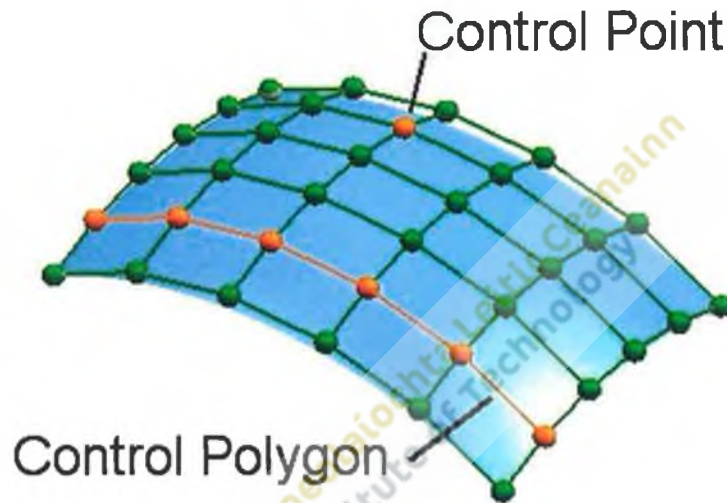


Figure 2-5 NURBS control point and control polygon

2.2.2.2 Hierarchical Model

Geometric models often have a hierarchical structure made through a bottom-up construction process. Components are used as building blocks to create higher-level entities. “Object hierarchies are common because almost all entities are divisible and at least a two-level hierarchy. In the uncommon case that each object is included only once in a higher-level object, the hierarchy can be symbolized as a tree, with objects as nodes and inclusion relations between objects as edges”(Foley 1997).

Tree style hierarchical models are often used in character animation. As the model shown in Figure 2-6, character is described by joints connected by rigid length in a hierarchy. Degree of freedom (DOF) specifies the number of motion channels. Most human joints have one or two DOFs but some of them have three. For example, the head joint is usually modelled with three DOFs, but the foot joint is usually modelled with only two DOFs and toe joint normally modelled with one DOF. The root of the skeleton normally has three translational DOFs and three rotational DOFs. Figure 2-7 demonstrates the details of tree-structure skeleton.

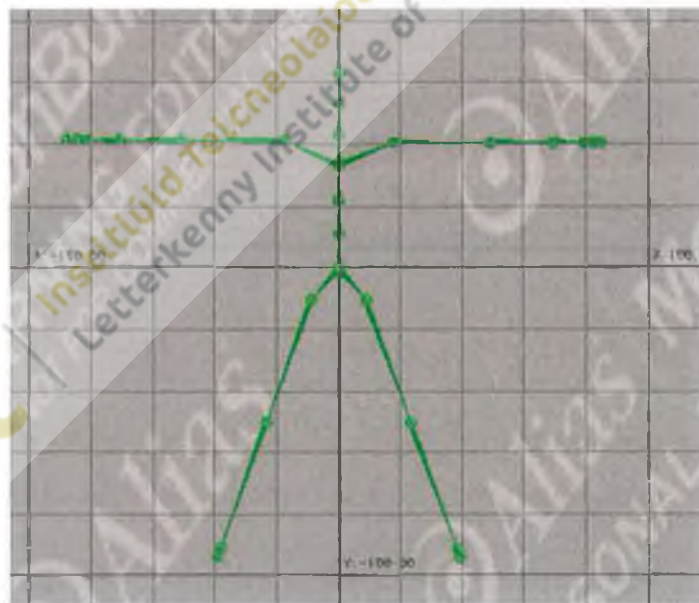


Figure 2-6 Skeleton in Front View

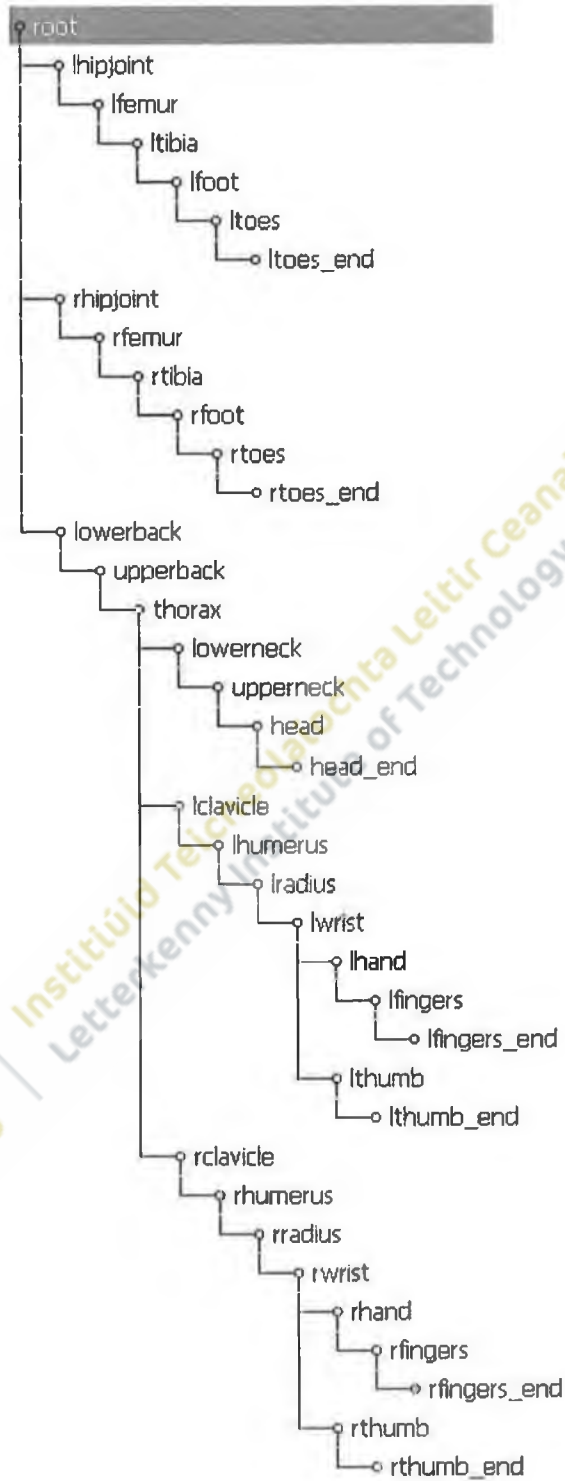


Figure 2-7 Tree structure of skeleton

2.2.3 Motion Description Methods

In general, motion can be described or generated by following three methods: keyframing, motion capture and procedural method. In keyframing, the animator specifies key values for the animated DOFs and the computer interpolates between these values. Motion capture is the process of recording motions of an actor and mapping them to a model. Procedural methods identify motions algorithmically.

2.2.3.1 Keyframing

Keyframing is the simplest and oldest form of animating an object. It is based on the notion that an object has a beginning state or condition and will be changing over time, in position, form, colour, luminosity, or any other property, to some different final form. Key frames are the frames in which the entities being animated have extreme or characteristic positions, and relies on all other intermediate positions being calculated consequently. (Foley 1997) Applications have been developed to provide graphical interfaces for animators to model, animate, and render the animation. Keyframing gives the animator a good control of the animation. However, it requires intense labour to generate a completed product and is thus very time consuming. For example, the animated film Shrek 1 (May 2001 DreamWorks Animation) took almost three years to produce.

2.2.3.2 Procedural Animation

To automatically generate animation in real-time allows generating more actions than could be created using predefined animations. Procedural animation is used to simulate particle systems, cloth and clothing, rigid body dynamics, and hair and fur dynamics, as well as character animation. Very realistic effects can be generated that would very hardly be possible with traditional animation. After a control system is built, the user can create animation by giving high-level commands. In addition, dynamic simulated clothing, hair, and muscle and their interaction with the surfaces of the figure contribute significantly to the character. However, procedural animation is the animation generated by a complex system and it requires enormous human and finance support.

2.2.3.3 Motion Capture

Alberto Menache (Menache 1999) said that motion capture is "The creation of a 3D representation of a live performance." It is a fairly controversial tool for creating animation. A motion capture session records only the movements of the actor, not visual appearance. These movements are recorded as animation data which later are mapped to a 3D model (human, robot, etc.). Motion data allows the model which is created by a computer artist to move the same way as the actor.

Motion capture offers several advantages over traditional computer animation of a 3D model. Motion data can be obtained rapidly; sometimes even real time results can be achieved. The level of complexity is stable even when the length of the

performance various comparing to the traditional methods. Complex movement and realistic physical interactions can be more easily recreated in a physically accurate style. Motion capture technology allows one actor to play multiple roles within a single film.

The director can choose any camera angle desired for a scene, including angles that are difficult or impossible to film in live action situations. Costumes, make-up, body size and age can be changed to whatever is needed. The characters will blend perfectly in with their digital environments. There is no need to have light, colours and filters in mind when filming the motions, as these can be added digitally later.

Afterwards the data can be manipulated in many ways to improve the quality or include elements not present in the original.

At the same time, disadvantages of motion capture are obvious. The high cost of the software and equipment and personnel required can be prohibitive for small productions. The contents of the data are limited to what can be performed without extra editing of the data. Movement that does not follow the laws of physics generally cannot be represented. The real life performance may not translate on to the computer model as expected. It is sometimes easier to re-shoot the scene rather than trying to manipulate the data. Only a few systems allow real time viewing of the data to determine if there is a necessity of redoing the motion.

In a motion capture system, a performer wears markers that are tracked. The motion capture computer software records the positions, angles, velocities, accelerations, impulses and other data, and providing an accurate digital representation of the motion. Markers are placed near each joint to identify the positions or angles between them. Markers may be acoustic, inertial, LED, magnetic or reflective markers, or combinations of any of these.

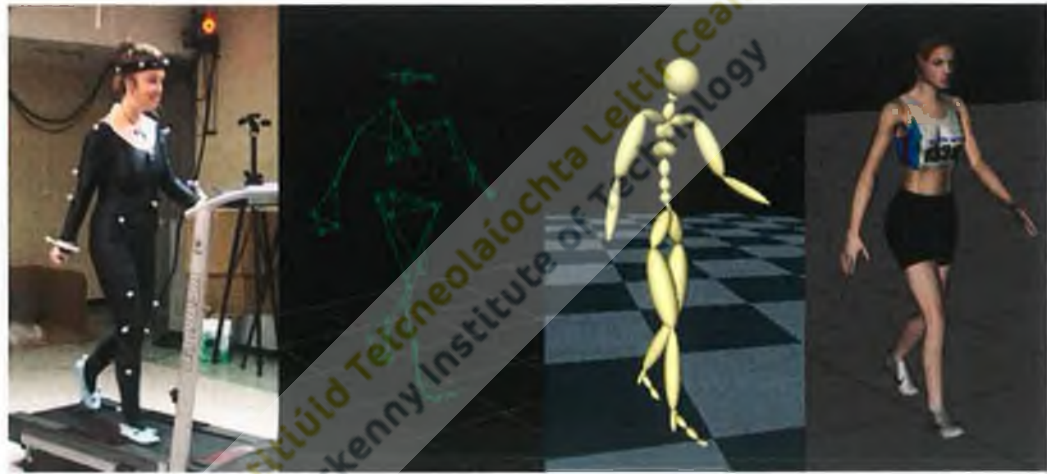


Figure 2-8 Motion capture system by

http://vrlab.epfl.ch/research/LO_locomotin_engine.html

There are two ways of getting motion data; either use a motion capture system to generate one or use already defined data. In Figure 2-8, walking motion is captured by motion capture system and then (or in real-time) the walking motion is simulated by the simulation software. Motion capture systems are expensive. On the other hand, the advantages are obvious as well. It is easier to get motions with all kinds of specifications because to modify a motion data is much more difficult

than to capture a new motion from the start. Leading companies like Vicon, Animazoo and Moven provide high-end products. The cost of the product depends on the specifications of the system. Below is a list of costs of the major products of motion capture system with foundation package.

<i>Brand</i>	<i>Cost</i>
<i>VICON</i>	<i>\$ 250,000</i>
<i>Moven</i>	<i>\$ 50,000</i>
<i>Gypsy</i>	<i>\$ 35, 000</i>
<i>3dsuit</i>	<i>\$ 25,000</i>

Table 2-1 Price comparison of motion capture system on the market

In Los Angeles January 22, 2008, VICON announced its low-cost motion capture system- FK Extreme, which costs approximately \$50,000. Last year a company called NaturalPoint announced their motion capture system only cost \$5000. The quality of the captured data is lower than Vicon's standard suit (a 48 camera MX40 motion capture solution). But for small studios and private users it is affordable and the result is reasonable.

There are a few institutes that provide motion capture library to everybody free of charge, for example, Carnegie Mellon University (CMU) Graphics Lab Motion Capture Database, StockMoves, mocapdata.com, and truebones. The advantage of such data is that it is free. The disadvantage is that data needs to be modified

individually according to every particular case. It appears there is a consensus amongst the researchers and technicians in the area that editing motion data is more difficult than generating one using motion capture systems. The motion data used in this project are from Carnegie Mellon University (CMU) Graphics Lab Motion Capture Database. CMU motion capture lab uses Vicon cameras to catch actor's movement in a local space, then the Vicon software system called "ViconIQ" processes the camera data which roughly includes using Vicon skeleton template create skeleton, label markers and export skeleton file.

2.3 Mathematics

The configuration of a skeleton of any movement is described by the orientation of its bones. The orientation of a bone (line segment) can be described by the so-called Euler Angle.

2.3.1 Euler Angles

The Euler angles describe the orientation of a rigid body (a body in which the relative position of all its points is constant) in 3-dimensional Euclidean space. To give an object a specific orientation it may be subjected to a sequence of three rotations described by the Euler angles. This is equivalent to saying that a rotation matrix can be decomposed as a product of three elemental rotations.

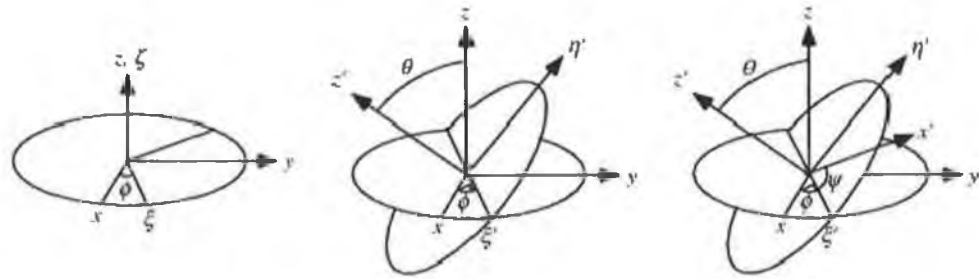


Figure 2-9 Euler Angle from Wolfram Mathworld

According to Euler's rotation theorem, any rotation may be described using three angles. If the rotations are written in terms of rotation matrices B, C, and D, then a general rotation A can be written as:

$$A = BCD \quad [\text{Equation 2-5}]$$

The three angles giving the three rotation matrices are called Euler angles. There are several conventions for Euler angles, depending on the axes about which the rotations are carried out.

The so-called "x-convention," illustrated in Figure 2-9, is the most common definition. In this convention, the rotation given by Euler angles (ϕ, θ, ψ) , where the first rotation is by an angle θ about the z-axis, the second is by an angle $\phi \in [0, \pi]$ about the x-axis, and the third is by an angle ψ about the z-axis. Euler angles (ϕ, θ, ψ) are called differently in different areas. In airplane industry they called heading, attitude and bank and their angular velocities are called roll, pitch and yaw.

In the x -convention, the component rotations are then given by:

$$D \equiv \begin{bmatrix} \cos \phi & \sin \phi & 0 \\ -\sin \phi & \cos \phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad [\text{Equation 2-6}]$$

$$C \equiv \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & \sin \theta \\ 0 & -\sin \theta & \cos \theta \end{bmatrix} \quad [\text{Equation 2-7}]$$

$$B \equiv \begin{bmatrix} \cos \psi & \sin \psi & 0 \\ -\sin \psi & \cos \psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad [\text{Equation 2-8}]$$

Euler angles have some drawbacks. Firstly, a single rotation can be represented by several different sets of Euler angles. Then, a so-called “gimbal-lock” can occur due to the order in which the rotations are performed. Joint angles show large variations when the robot is near gimbal-lock. Regions near gimbal-lock were encountered frequently in these motions because the robot shoulder has a singularity when the arms are at 90 degrees abduction, or swung out to the side of the body to a horizontal position. In this position for the humeral rotation, one degree-of-freedom is lost. There are many ways to address this problem. One of the methods is given by Nancy et al (Nancy S. Pollard 2002). When a joint angle’s

range is near gimbal-lock an assumption is given by interpolating between start and end of the time period. Then a joint angle solution is computed assuming the robot is in a singular configuration. Another way to avoid gimbal-lock is by using quaternions.

2.3.2 Quaternion

Quaternion is another representation of rotations. The quaternions were first invented (discovered) by Irish mathematician Sir William Rowan Hamilton (Hamilton 1844). It is defined as:

$$H = \{ (a, v) \mid a \in \mathbb{R}, v \in \mathbb{R}^3 \} \quad [\text{Equation 2-9}]$$

A quaternion is now often interpreted as (a, v) where a is a real number and v is a 3D vector. Quaternion addition is defined by:

$$q_1 + q_2 = (a_1 + a_2, v_1 + v_2) \quad [\text{Equation 2-10}]$$

Quaternion multiplication is defined as:

$$q_1 \bullet q_2 = (a_1, v_1) \bullet (a_2, v_2) = (a_1 a_2 - \langle v_1, v_2 \rangle, a_1 v_2 + a_2 v_1 + v_1 \times v_2)$$

$$[\text{Equation 2-11}]$$

Where $\langle v_1, v_2 \rangle$ denotes the standard scalar product of two vectors and \times denotes the cross product. Quaternion multiplication is not commutative but is associative. The quaternion:

$$q = [1, (0, 0, 0)] \quad \text{[Equation 2-12]}$$

is the multiplicative identity. The addition identity quaternion is:

$$q = [0, (0, 0, 0)] \quad \text{[Equation 2-13]}$$

A unit quaternion (a, v) satisfies:

$$a^2 + v_1^2 + v_2^2 + v_3^2 = 1 \quad \text{[Equation 2-14]}$$

2.3.2.1 Angle between Two Quaternions

A quaternion can be considered as a 4-D vector. The angle between two quaternions can be computed using the inner product as below:

$$\theta = \arccos\left(\frac{\langle q_1, q_2 \rangle}{\|q_1\| \cdot \|q_2\|}\right) \quad \text{[Equation 2-15]}$$

$\|q\|$ denotes the norm of the quaternion

$$\|q\| = \sqrt{a^2 + v_1^2 + v_2^2 + v_3^2}$$

[Equation 2-16]

2.3.2.2 Advantages of Quaternion

The use of quaternions has advantages over the use of rotations matrices in many situations. They avoid the problem of "gimbal-lock". Instead of rotating an object through a series of successive rotations about mutually perpendicular directions, quaternions allow the programmer to rotate an object through an arbitrary rotation axis and angle. Quaternions require less storage space. Concatenation of quaternion requires fewer arithmetic operations, quaternions are more easily interpolated for producing smooth animation.

2.3.2.3 Conversion between Euler Angle and Quaternion

An Euler Angle can be described as:

(heading, attitude, bank)

A quaternion q can be described as:

$$q = (a, v) = (a, (v_1, v_2, v_3))$$

Convert Euler Angle to Quaternion:

$$a = c_1 c_2 c_3 - s_1 s_2 s_3$$

$$v_1 = s_1 s_2 c_3 + c_1 c_2 s_3$$

$$v_2 = s_1 c_2 c_3 + c_1 s_2 s_3$$

$$v_3 = c_1 s_2 c_3 + s_1 c_2 s_3$$

Where:

$$c_1 = \cos\left(\frac{\text{heading}}{2}\right)$$

$$c_2 = \cos\left(\frac{\text{attitude}}{2}\right)$$

$$c_3 = \cos\left(\frac{\text{bank}}{2}\right)$$

$$s_1 = \sin\left(\frac{\text{heading}}{2}\right)$$

$$s_2 = \sin\left(\frac{\text{attitude}}{2}\right)$$

$$s_3 = \sin\left(\frac{\text{bank}}{2}\right)$$

Convert Quaternion to Euler Angle:

$$\text{heading} = \arctan\left(\frac{2 \cdot v_2 \cdot a - 2 \cdot v_1 \cdot v_3}{1 - 2 \cdot v_2^2 - 2 \cdot v_3^2}\right)$$

$$\text{attitude} = \arcsin(2 \cdot v_1 \cdot v_2 + 2 \cdot v_3 \cdot a)$$

$$\text{bank} = \arctan\left(\frac{2 \cdot v_1 \cdot a - 2 \cdot v_2 \cdot v_3}{1 - 2 \cdot v_1^2 - 2 \cdot v_3^2}\right)$$

Except when $v_1 \cdot v_2 + v_3 \cdot a = 0.5$

$$\text{heading} = 2 \cdot \arctan\left(\frac{v_1}{a}\right)$$

$$\text{bank} = 0$$

And when $v_1 \cdot v_2 + v_3 \cdot a = -0.5$

$$\text{heading} = -2 \cdot \arctan\left(\frac{v_1}{a}\right)$$

$$\text{bank} = 0$$

2.3.2.4 LERP and SLERP

Interpolation is a technique that helps to generate the intermediate frames between key frames. Linear interpolation (LERP) is the simplest type of interpolation. For two unit quaternions q_1 and q_2 , the interpolated quaternion $q(t)$ is given by:

$$q(t) = (1-t)q_1 + tq_2 \quad [\text{Equation 2-17}]$$

The quaternion $q(t)$ changes as t varies from 0 to 1 and it ends at line AB. See Figure 2-10.

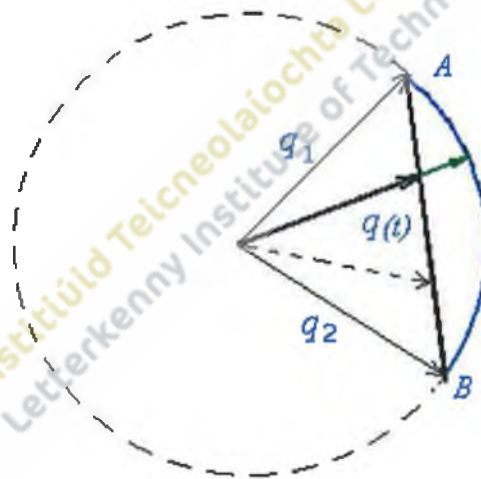


Figure 2-10 Linear Interpolation

The quaternion $q(t)$ does not maintain the unit length of q_1 and q_2 . Equation 2-17 can be changed by normalizing both sides. Then:

$$q(t) = \frac{(1-t)q_1 + tq_2}{\|(1-t)q_1 + tq_2\|} \quad [\text{Equation 2-18}]$$

This is the function represented by $q(t)$ that followed the arc AB.

LERP is very efficient because it needs very few calculations. However, it has a drawback; when t varies from 0 to 1 the rate of change is not even. In other words, the angle between $q(t)$ and q_1 (or $q(t)$ and q_2) does not change smoothly. In fact, when $t=0.5$, the change is the fastest and when at the end points ($t=0$ and $t=1$), the change is the slowest. Spherical Linear Interpolation (SLERP) solves this problem. Angle θt and angle $\theta(1-t)$ represent the angle between q_1 and q_2 .

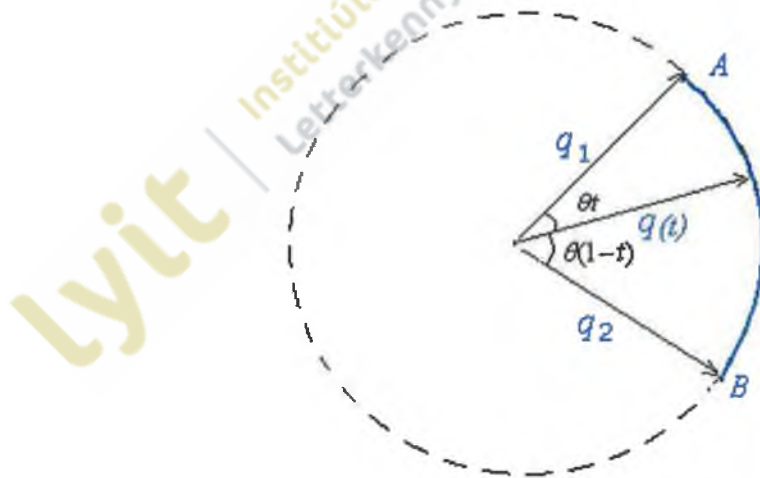


Figure2-11 Spherical Linear Interpolation

According to the relationship of trigonometric $q(t)$ can be represented as following:

$$q(t) = \frac{\sin \theta(1-t)}{\sin \theta} q_1 + \frac{\sin \theta t}{\sin \theta} q_2 \quad [\text{Equation 2-19}]$$

Angle $\theta = \cos^{-1}(q_1 \cdot q_2)$ thus $\sin \theta = \sqrt{1 - (q_1 \cdot q_2)^2}$. In order to ensure that the interpolation takes place over the shortest path quaternion q_1 and q_2 usually qualifies that $q_1 \cdot q_2 \geq 0$ because quaternions q and $-q$ represent the same rotation.

2.3.3 Inverse Kinematics

The common use of inverse kinematics is to make sure characters connect physically to the world, such as feet landing firmly on top of terrain. An articulated figure consists of a set of rigid segments connected with joints. Altering angles of the joints can cause an enormous number of configurations. Kinematics consists of two types – forward kinematics and inverse kinematics. Forward kinematics is the process of calculating the position in space of the end of a linked structure by given the angles of all the joints. Inverse kinematics (IK) does the reverse. Given the end point of the structure, IK is the process of finding the joint angles in the linked structure. The solutions can be difficult and usually can be many or infinitely many. Inverse kinematics is an essential part of many motion editing techniques.

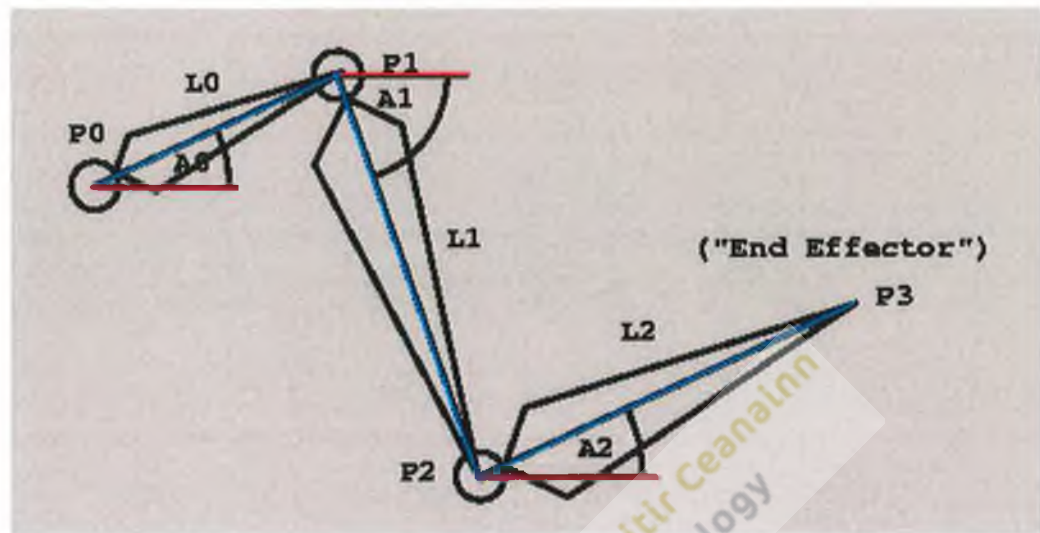


Figure 2-12 Inverse Kinematics Mechanism by Stewart Dickson in "Digital Character Construction"

In the Figure 2-12, Forward Kinematics (in 2D) is that given A_0 , A_1 , A_2 , L_0 , L_1 , L_2 and start position P_0 solve end position P_3 . Inverse Kinematics is that given L_0 , L_1 , L_2 and start position P_0 and end position P_3 solve A_0 , A_1 and A_2 .

IK can be useful in real-time animation, for example the user throws a ball, and the computer moves its player's glove to catch it; or a character is dropped onto random uneven terrain, and keeps his feet at different heights to stand or walk on the terrain; or a character is walking down stairs, and automatically moves his hand to hold onto the railing, and moves his feet to adjust to different sized stairs.

Motion blending is often used to generate motion transitions, but blending may introduce artefacts into the resulting motion. One common scenario is that the

character's feet move when they are supposed to remain planted, this artefact is normally called foot-slide, see Figure 2-13. Inverse kinematics is introduced to solve this problem. Rose et al.(Rose 1996) used this approach to handle space-time constraints and inverse kinematics constraints. Their method for creating motion transitions uses a combination of these two constraints in order to generate seamless transitions.

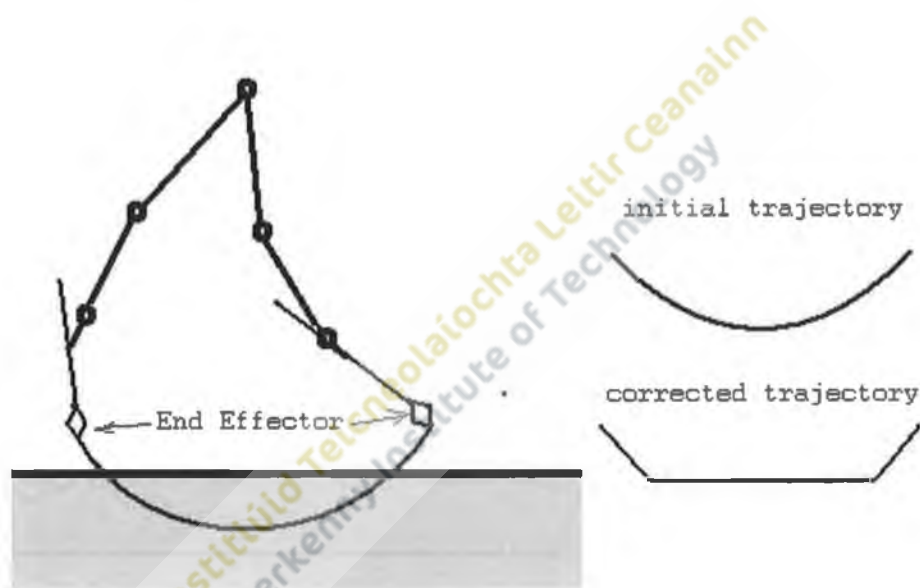


Figure 2-13 Foot-slide

Kovar et al.(Kovar 2002) presented an algorithm for removing foot-slide artefacts introduced by motion capture editing. After using an analytic IK algorithm, smooth adjustments to skeletal parameters is added when trying to satisfy constraints. Inverse kinematics has been used primarily to position limbs to maintain constraints in motion editing systems. Inverse kinematics is then used as a post process to fix foot-slide introduced by linear blending. Many algorithms

have been proposed in previous works to address these artefacts, see (Bruderlin 1995; Witkin 1995; Rose 1998; Rose 2001; Wang 2004; Ball 2008). The inverse kinematics solver provided by MotionBuilder 7.0 is used to constrain support limbs and correct foot-slide.

2.3.4 Catmull-Rom Splines

Catmull-Rom splines are a set of cubic interpolating splines with the tangent at each point calculated by using the previous and next point on the spline. Catmull-Rom splines can be defined by a simple process of deduction as follows, see (Catmull 1974).

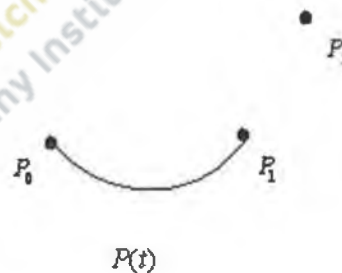


Figure 2-14 Cubic Curves

$P(t)$ is a cubic curve with the condition that when $t = 0$, $P(t)$ yields to P_0 and when $t = 1$, $P(t)$ yields to P_1 . The general equation of cubic curve is:

$$P(t) = at^3 + bt^2 + ct + d \quad \text{[Equation 2-20]}$$

P_{-1} and P_2 adjust the shape of the curve by defining the tangent of two points as following:

$$\begin{aligned} P'(0) &= \alpha(P_1 - P_{-1}) \\ P'(1) &= \alpha(P_2 - P_0) \end{aligned} \quad \text{[Equation 2-21]}$$



Figure 2-15 Catmull-Rom Spline

Note that the tangent at point P_{-1} is not clearly defined. It is often set to $\alpha(P_0 - P_{-1})$. α changes from 0 to 1, $P(0) = 0$, $P(1) = 1$. It yields:

$$\begin{bmatrix} P_0 \\ P_1 \\ P'(0) \\ P'(1) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\alpha & 0 & \alpha & 0 \\ 0 & -\alpha & 0 & \alpha \end{bmatrix} \begin{bmatrix} P_{-1} \\ P_0 \\ P_1 \\ P_2 \end{bmatrix} \quad \text{[Equation 2-22]}$$

And since $P'(t) = 3at + 2b + c$

$$\begin{aligned}
 P(0) &= d \\
 P(1) &= a + b + c + d \\
 P'(0) &= c \\
 P'(1) &= 3a + 2b + c
 \end{aligned}
 \tag{Equation 2-23}$$

In matrix:

$$\begin{bmatrix} P_0 \\ P_1 \\ P'(0) \\ P'(1) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}
 \tag{Equation 2-24}$$

From Equation 2-23 and Equation 2-24 it has:

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\alpha & 0 & \alpha & 0 \\ 0 & -\alpha & 0 & \alpha \end{bmatrix} \begin{bmatrix} P_{-1} \\ P_0 \\ P_1 \\ P_2 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}
 \tag{Equation 2-25}$$

And then:

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix}^{-1} \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\alpha & 0 & \alpha & 0 \\ 0 & -\alpha & 0 & \alpha \end{bmatrix} \quad [\text{Equation 2-26}]$$

After matrices transformation:

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} -\alpha & 2 - \alpha & \alpha - 2 & \alpha \\ 2\alpha & \alpha - 3 & 3 - 2\alpha & -\alpha \\ -\alpha & 0 & \alpha & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad [\text{Equation 2-27}]$$

Put Equation 2-27 into standard matrix equation:

$$P(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -\alpha & 2 - \alpha & \alpha - 2 & \alpha \\ 2\alpha & \alpha - 3 & 3 - 2\alpha & -\alpha \\ -\alpha & 0 & \alpha & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_{-1} \\ P_0 \\ P_1 \\ P_2 \end{bmatrix} \quad [\text{Equation 2-28}]$$

In equation 2-28, $P(t)$ actually represents another type of splines called Cardinal splines. Parameter α represents spline's 'tension'. Tension α changes from 0 to 1.

When tension approaches 1 the bend at each knot is less, as if the spline is a rope.

A tension value of $\alpha = \frac{1}{2}$ is commonly used to represent Catmull-Rom spline.

$$P(t) = \frac{1}{2} \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_{-1} \\ P_0 \\ P_1 \\ P_2 \end{bmatrix} \quad [\text{Equation 2-29}]$$

Equation 2-29 gives Catmull-Rom spline certain characteristics.

- The spline passes through all of the control points.
- The spline is C1 continuous, meaning that there are no discontinuities in the tangent direction and magnitude.
- The spline is not C2 continuous. The second derivative is linearly interpolated within each segment.

Catmull–Rom splines are often used to get smooth interpolated motion between key frames. For example, most camera path animations generated from discrete key-frames are handled using Catmull–Rom splines.

2.4 Related Works

2.4.1 Motion Blending and Warping

Motion blending and warping consist of a widely accepted collection of standard techniques, which allow generation of new motions by interpolation between motion-captured sequences. In most cases, every set of motion capture data only

contains a series of specific actions for some purpose, e.g. walking or jumping. To generate a more complicated action, e.g. running after walking, two or more sets of original motion capture data need to be smoothly connected by a certain technique, which is called “motion blending and warping”.

Guo et al.(Guo 1996) and Wiley et al.(Wiley 1997) provided interpolation techniques for example motion located regularly in parameter spaces. They produce new motion using linear interpolation on a set of example motions. Rose et al.(Rose 1996) generate transitions using a combination of space-time constraints and inverse kinematics constraints to create dynamically a plausible transitions mechanism and a verb graph to smooth motion transition. Sloan et al. (Sloan 2001) provided a more efficient scheme by reconstructing their formula.

Unuma et al.(Unuma 1995) use digital signal processing techniques to interpolate and extrapolate motion data. Amaya et al. (Amaya 1996) alter existing animation by extracting an “emotional transform” from example motions which is then applied to other motions. These early methods can not handle non-periodic motions.

Bruderlin and Williams (Bruderlin 1995) use multi-target interpolation with dynamic time warping to blend between motions, and displacement mappings to alter motions such as grasps. Witkin and Popovic (Witkin 1995) present a similar system for editing motion capture clips. Perlin (Perlin 1995) has approached this

problem in a very different way by using noise functions to simulate personality and emotion in existing animation.

Other schemes generate realistic motion by choosing postures in captured motions while traversing the motion graphs representing the examples. Posture rearrangement generates a motion by seamlessly rearranging existing postures in example motions.

Kovar et al. (Kovar 2002) pointed out that automatically generating a convincing transition is as difficult as creating a new motion in the first place after they introduced a motion graph to represent the transitions among the poses. Other researches (Lee 2002) present a similar motion graph method and provide user interface for interactive motion control. Arikan et al. (Arikan 2002) developed a method for satisfying user-specified annotations while rearranging motion segments. Latterly Reitsma et al. (Reitsma 2007) created methods that can be used to evaluate the extent to which a motion graph will fulfil the requirements of a particular application, lessening the risk of the data structure performing poorly at a bad time.

These schemes can generate realistic motions. However, they need large amounts of calculation especially when large numbers of example motions involved.

2.4.2 Motion Transition

Transitions are an essential component, but the emphasis of most researches (Kovar 2002; Lee 2002; Sidenbladh 2002) was on selecting appropriate transition points rather than the durations of transitions. Transitions are less important if the motions are similar, visual artefacts can still emerge if the duration is not properly adjusted.

Most researchers either use start-end method (Rose 1996; Rose 1998; Kovar 2002) or centre-aligned transition specification (Arikan 2002; Kovar 2003) Pullen and Bregler (Pullen 2000) join motions directly, but then blend the motion with a smooth quadratic fitted to the curves. Jing Wang and Bobby Bodenheimer (Wang 2004; Wang 2008) analyzed the advantages and disadvantages of these methods. Start and end frames have the advantage that they are intuitive and easy to specify. They also work well if the transition points are at the end or beginning of motion segments. Their disadvantage is that they can change the alignment of the motions as they are changed. Centre-aligned transitions have fixed alignment, which are both an advantage and a disadvantage. If the centre-aligned poses are quite similar, then a centre-aligned transition is more robust to variations in the blend length. On the other hand, if the poses are mismatched, then blending will not make the transition look good. Centre-aligned transitions also have the disadvantage that depending on the blend length there is a region at the beginning and end of each motion segment for which a true blended transition cannot be made. Wang and Bodenheimer (Wang 2003; Wang 2004) chose to specify transitions with start and

end frames. At present centre-aligned transitions rely too heavily on transition metrics. Wang and Bodenheimer also noted that different motions behave differently under these transitions metrics, tuning is required, and there are no guarantees that an optimal transition selected by a method is visually appealing. Therefore, changing the transition points by changing the alignment, if it can be done in a computationally efficient way, represents a second-pass process that can improve the visual appeal of a transition.

2.5 Conclusion

Animation is often perceived as an important technology and has the potential to improve many aspects of life and the economy including entertainment, education, health and employment.

Alongside the traditional animation techniques, computer-generated animation has been developed into a mainstream technique when the computers aided animation development.

Computer-generated animation focuses on objects in change. Objects are described in 3D modelling techniques, where most popular primitives are polygons and splines, especially NURBs. The structure of the model is important for manipulating models. As to geometrical model, the hierarchical structure or the tree structures are most commonly used.

Motions can be described by methods including key-framing, procedural method and motion capture. In order to improve productivity and cut the cost, reconstructing the existing motion captured data is a good solution.

The development of these methods involves a strong mathematical background. Quaternions and Euler angles are used to define configurations in 3D. In order to connect two motions, some interpolating methods need to be introduced. Commonly used methods include linear interpolation (LERP) and Spherical Linear Interpolation (SLERP).

Such developed motions would generally experience some artefacts. In order to remove artefacts, such as foot skating, the technique of inverse kinematics can be used. The artefacts, such as the inconsistency of joint movement may be smoothed using Catmull-Rom spline.

The investigation of related studies conducted by other researchers demonstrated the overall picture of the academic achievement in the motion blending and warping area. Inspiration was drawn from other people's work, such as Witkin and Popovic's blending between two motions sequences using an efficient system and Wang and Bodenheimer's approach to motion transition.

In the next chapter the research methodology is considered.

3. Research Methodology

3.1 Introduction

Following the literature review section, which has provided the reader with an in-depth understanding of motion transition, the study now focuses on the methods utilised as part of this research.

This chapter describes the development of the project. There are many research methods available, and this chapter outlines these research methods and justifies the particular method that was chosen for this study. The research method chosen can depend on various factors, such as time availability, the particular area that is researched and the individual research style.

3.2 General Research Method

The general method for the entire research can be described as “reading – analysing – developing”. First of all, the subject matter needs to be identified. The correct concepts and proper knowledge related to the subject should be studied.. Then the scope of study should be narrowed down and particular problems targeted and analysed. This is a very important step and takes the longest time. A small mistake can cause the failure of the development of the entire project.

Finally, after the problem has been examined and analysed the proposed the solution can be generated and , the actual development can commence.

The development involved many stages. Based on the complexity of the software, time, number of programmers and other factors a software development method can be chosen. In case of this research the waterfall model was selected. The standard waterfall model for software management includes following stages:

1. Requirements specification
2. Design
3. Construction (implementation or coding)
4. Integration
5. Testing and debugging (Validation)
6. Installation
7. Maintenance

The last two stages were deemed not suitable for this project. Therefore, a five-stage waterfall model was used to manage the development.

Above section explains general research methodology. As part of the development, it was decided that certain additional data needed to be collected in order to apply most suitable blending length in the clip. The research methods considered for this task are outlined below.

Each of the research strategies, such as experiment, survey, action research and case study, uses specific data collection instruments and has different benefits. An appropriate strategy is based on identified research question, objectives and available resources.

3.2.1 Survey

The survey strategy is particularly suited for describing characteristics of a large population and, with the combination of a carefully selected sample and a standardised questionnaire, a survey can offer the possibility of making refined descriptive assertions about a student body, a city, a nation or other large population (Babbie 2001).

By using the survey strategy, statistical analysis tools can be used to process the data collected from a survey, e.g. SPSS. However, in order for the statistic analysis to yield valid results, the data must be first prepared, with quantitative analyses in mind and having considered when to use different charting and statistical techniques (Saunders 2003).

It was decided that the survey method was most suitable method to produce useful and informative data about blend length as it allowed obtaining input from a number of respondents. The detailed rationale and description of the utilised strategy is further explored in Chapter 4.

3.3 Data Collection Methods

There are multiple options available as to which data collection method should be used, such as questionnaire, observation and interview. Each data collection method is deemed suitable for different research studies.

3.3.1 Observation

The observation method is used for examination of behaviours; observation involves the systematic observation, recording and interpreting peoples behaviour (Saunders 2003). Observation can be carried out either qualitatively or quantitatively (Saunders 2003). There are varying degrees of participation ranging from the researcher's identity being revealed as either an observer or participant in the activity (Saunders 2003).

The second observation technique is structured observation and it is concerned with frequency of actions. Structured observation is a time consuming task but it is relatively easy to carry out and could be delegated to a few people to carry out on different locations, if a comparison study was needed.

Observation is not deemed an appropriate data collection technique for this study as it would not collect the required participants' commentary.

3.3.2 Interviews

An interview is characterised by a direct contact with the respondent, asking questions and recording answers. Interviews are particularly useful when dealing with complex issues and when open questions are used to collect data. According to Cooper and Schindler (Cooper 1998) the main advantage of interviews over other techniques, such as questionnaires, is the depth and detailed information they provide.

In this study there is no necessity of obtaining detailed information; therefore, interview is not a suitable data collection technique.

3.3.3 Questionnaire

The survey strategy makes the greatest use of this technique. (Saunders 2003) outlines that questionnaires can be an efficient data collection method when the required information has been identified; it is usually administered to a large sample of respondents. Questionnaires are sometimes accompanied by interviews, which are used to gain an in-depth understanding to the responses to the questionnaire (Saunders 2003). There are different types of questionnaires including; on line, postal, in person (on the street) and telephone questionnaires.

Questionnaires are usually not suited for exploratory research as exploratory research requires more open ended questions. A great deal of thought needs to go

into the design of the questionnaire in order to yield reliable and valid data and that the desired data is collected to answer the research questions (Saunders 2003).

Questionnaire is deemed most suitable data collection method for the identified matter as it will allow obtaining input from a number of respondents, preferably users of similar applications in order to determine the most suitable blending length. The detailed rationale and description of the utilised data collection method is further explored in Chapter 4.

3.4 Conclusion

The research focuses on the development of a novel approach based on existing known techniques. The research is analytic and exploratory. The method “reading – analysing – developing” is used to conduct the research. In the section on choosing blending length, a survey is the method chosen along with a questionnaire as the data collection technique.

4. Design

4.1 Introduction

This chapter discusses the process of develop a method to connect two motion sequences to create a new realistic natural looking motion sequence.

4.2 Real-time Animation

Real-time animation is that process allows animation to be generated or constructed in real time. Computer games and other similar interactive software applications are the major area of real-time animation. In game development animation is the major factor that makes a game graphically pleasing and attractive. Lots of researches about animation have been carried out and the results are convincing. However, most of the methods are complex and require an extra input which does not fulfill the requirement of real-time animation.

In a real-time situation, a game player presses a button or a key to let the character run whilst the character is walking. Behind the scene game program invoke a method with the motion database and other pre-processed data available generates a real-time animation, and then displays it on the screen. In game development an animator normally is the person taking care of animation. The number of players in a game at the same time can be thousands or more. The game world is so vast and complicated. To animate four basic actions and to transit between each other

in advance is not an unachievable job. However with forty different types of motions that is an unbelievable work load. In such a case a project director would have to hire many more animators to do the work and this would increase project costs.

Thanks to the rapid development in the computer industry, high speed processors and big storage space are available. It is essential that the game server has a fast processor and parallel computing ability in order to apply real-time animation because real-time animation the increases real-time calculation of the processor. Real-time animation demands a method that can apply to general motions to create a new motion. This method should be efficient and simple.

4.3 Visualisation Tool

The visualization tool used in this project is provided by Carnegie Mellon University Motion Capture library project. It is revised twice by Steve Lin, Alla and Kiran Janery 2002. It is originally designed for the convenience of beginners in the area of 3D graphics programming. Unlike Motion Builder, Maya or 3D Max Studio, It is not an industrial standard 3D visualization tool. This tool is an open source tool and is available for everyone to use. The data was in convenient form for the tool.



Figure 4-1 3D Viewer Interface

As Figure 4-1 shows, the Acclaim file player has a well structured interface. The interface tries to perform two major functions; displaying 3D model and playing an animation. There are also other functionalities such as change of light and background, change of 3D position, rotation of the model etc. This program is written in C++. Visualisation is a very important part in 3D graphics. A deeper understanding is provided by reading the source code. Fast Light Toolkit (FLTK) GUI is used as 3D API in the viewer. FLTK is a cross-platform C++ GUI toolkit for UNIX®/Linux® (X11), Microsoft® Windows®, and MacOS® X. FLTK

provides modern GUI functionality without being oversized and supports 3D graphics via OpenGL® and its built-in GLUT emulation. FLTK is designed to be small and modular enough to be statically linked, but works fine as a shared library. FLTK also includes an excellent UI builder called FLUID which can be used to create applications in minutes.

4.3.1 Data Format

The format is defined as the ASF/AMC. It is made up of two files, a skeleton file and a motion file. This was done knowing that most of the time a single skeleton works for many different motions and rather than storing the same skeleton in each of the motion files, it should be stored just once in another file. The skeleton file is the Acclaim Skeleton File (ASF). The motion file is the Acclaim Motion Capture data (AMC).

Acclaim is a game company which has been researching the motion capture for gaming purposes. They developed their own methods for creating skeleton motion from optical tracker data and devised a file format for storing the skeleton data. Subsequently, this format description has been released into the public domain. Oxford Metrics, makers of the Vicon motion capture system, chose to use the Acclaim format as the output format of their software.

4.3.2 Open Source and Drawbacks

There are some drawbacks of 3D Viewer. Although, there are still many programs developed through Visual Studio 6.0, it is considered slightly out of date since Visual Studio.NET is popularly used as a C++ development environment at the moment. The version of FLTK used in program is FLTK1.10. The further development FLTK1.2, is unsuccessful. The FLTK2.1 is an entirely different API.

4.4 Motion Initialization

4.4.1 Introduction

Motion data used here are directly downloaded from a motion captured database. The original motion data, such as running, jumping, kicking etc. used in games or other applications are periodical but for the convenience and efficiency of processing data, the motion should be in a full cycle. Only then, if the full cycle motion is played repeatedly, the motion will appear natural despite the position.

The method of motion initialisation is based on joint angles. Each frame of motion represents a posture of the character; each posture is a static pose of the character.

It means that each frame has a unique set of joint angles. In order to get the full cycle of a motion type the least differences between postures should be defined.

The least difference of two postures is defined by choosing a random frame as the start frame and then comparing each frame to it. Below instructions can be followed:

1. choose a random frame (posture) as the start frame(posture)
2. loop all other frames to perform subtraction operation on joint angles
3. combine all the joint angle differences to get an overall difference value
4. set the first chosen posture difference to zero
5. choose the smallest value of all postures
6. set the constraints to get rid of faulty frame to get the most similar frame to the original random frame.

The result of initialization is the set of motion data. The advantage of this data set is that each posture in the motion of this data set is unique, which makes it easier to operate. Initialized data set can now be further manipulated more effectively.

4.5 Blending and Warping

4.5.1 Blending

A very popular blending method used in recent researches is linear interpolation. Linear interpolation is efficient and easy to control. It is vital that the interpolating motion data presents the characteristics of both original motion and target motion. In this research the motion transition period is divided into five sections to connect the original motion and target motion. These six sections are called A (original like), B (close to original), C (convert original), D (convert target), E (close to target) and F (target like) as shown in Figure 4-2.

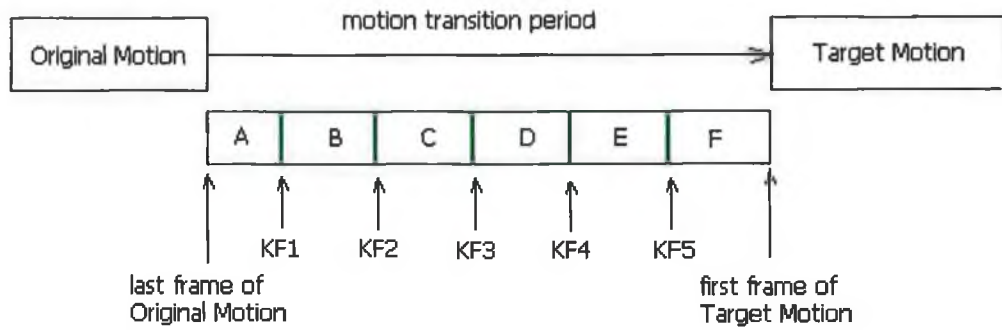


Figure 4-2 transition Segments

Each of these six sections is constructed by two key frames and in-between frames. For example, as presented in Figure 4-3, section A consists of key frames Original Motion frame (m-1) and Key Frame 1(KF1). In-between frames are generated via linear interpolation.



Figure 4-3 Key Frame Setups

All key frames, KF1, KF2, KF3, KF4 and KF5, are defined by the following equations:

$$\left(\begin{array}{l} KF1 = pAOM[osl] \\ KF2 = \frac{pAOM[osl] + pATM[nFrame - 3osl]}{2} \\ KF3 = \frac{pAOM[2osl] + pATM[nFrame - 2osl]}{2} \\ KF4 = pATM[2osl] \\ KF5 = pATM[osl] \end{array} \right) \quad \text{[Equation 4-1]}$$

In equation 4-1, pAOM represents a posture at original motion. pATM represents a posture at the target motion. osl represents one sixth length of blending length. One condition is that the original motion and the target motion have sufficient length.

4.5.2 Warping

Andrew Witkin and Zoran Popovic (Witkin 1995) describe a simple technique for editing captured or key framed animation based on warping of the motion. A set of constraints is used to derive a smooth deformation that preserves the characteristics of the original motion. The constraints include a set of angle-time pairs. Each angle must be assumed at the specified time. A similar approach to the one used by Witkin and Popovic was utilised in this research. Correspondences provide control points for a spline that controls the time map. Witkin and Popovic

select frames **manually** from a single motion sequence and modify key frames by using following model

$$\theta_{new}(t) = a(t)\theta_{old}(t) + b(t) \text{ [Equation 4-2]}$$

In Equation 4-2, $a(t)$ is used for scaling and $b(t)$ is used for offset. A new motion was specified by selecting a few frames from original and target motion according to certain rules specified in chapter 4.5. Then the spline was passed through the control points to get the result. It is important to note that this approach does not require manual interference from an animator.

Other research shows that it is better to interpolate between existing motions. for example, Kovar and Gleicher (Kovar 2003; Kovar 2004) developed a method which can automatically find similar clips of motion, automatically construct time-warps to synchronize, take weighted average to blend between them or solve for blend that achieves some user constraint. However, constraints and weights are complicated.

4.5.3 Position Issue

Root vectors are very important parameters in motion data. They determine the character's positions and body postures, which are the keys for a motion to be realistic-looking.

In this research, methods developed to calculate root positions are primarily considered to maintain consistency of motion flow. Facing direction is also an important issue, for example if a character in motion one faces south and the character in motion two faces north east, the transition to make this work need to turn the character anti-clock wise 135 degrees and then do rest of the transition. Turning a character means rotating the character around an arbitrary pole, which is a complicated and relatively independent technical issue. In this research the root issue is simplified by putting two motion sequences to a straight line and facing one direction.

The method used in this research to determine the position of the character is primarily based on the speed of both motion sequences. According to equations of motion in physics, positions can be calculated as follows:

$$P = P_0 + v_0 t + \frac{1}{2} a t^2 \quad [\text{Equation 4-3}]$$

$$v_2 = v_1 + a t \quad [\text{Equation 4-4}]$$

Equation [4-3] can be also illustrated as Figure 4-4, where P represents position in transition, P_0 represents start position of transition (which also can be understood as the last position of motion one), v_0 represents speed of motion one, t is the time, and a represents acceleration, which can be calculated according to Equation [4-4].

In Equation [4-4] both velocities v_1 and v_2 are known and t is the time cost in transition.

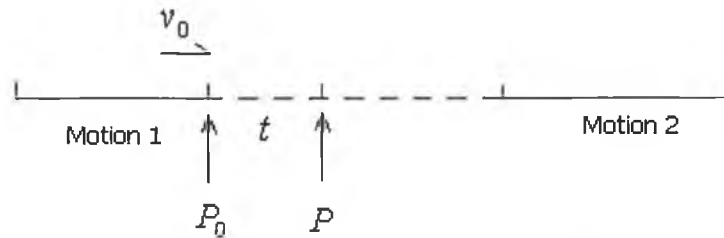


Figure 4-4 Root Position Calculation

4.6 Motion Transition Duration

4.6.1 Introduction

The duration of the transition is a critical factor in an animation stream. Rose et al. (Rose 1996) use transition durations of 0.3s to 0.8s. Lee et al. (Lee 2002) found a transition duration of 1 to 2 seconds worked well. Arikan and Forsyth (Arikan 2003) used a constant blend duration of 2 seconds. But they all left the exact specification of the duration to the operator.

Mizuguchi (Mizuguchi 2001) et al. were explicitly concerned with the blend length for transitions, but used an *ad hoc* method of determining them. In their experience, 10 frames (0.33s) worked for a wide variety of motions. Kovar et al. (Kovar 2002) also used this transition duration. None of the prior work attempted to compute an optimal duration for their particular method of transition generation.

Wang and Bodenheimer (Wang 2004) recognized the problem but considered it a confusing factor for their experiments and simply concatenated on motion segments. Wang and Bodenheimer (Wang 2004) used empirical methods to optimize the weights for computing transition points between motions.

This research is based on Wang and Bodenheimer's study admitting blending length is one of the key factors that can affect the quality of transition. It is assumed that the transition points are given and blend duration is the key factor to be decided.

An experimental method is used in this research instead of calculating duration through methods such as geodesic distance, joint velocity, Ad hoc comparison or others (Wang 2004). The experimental method decides the blend length according to the type of motion used.

4.6.2 Transition Duration Method

A method introduced in this research defines transition duration by categorising motion data types and joint speed with corresponding fixed blending length. The idea of constructing such a method is based on experiences and combined with physical principles.

Most of the recent researches use blending length from 0.3s to 2s. Commonly used rate of frames is 24 frames per second. In other words, the range of blends length is normally 8 to 48 frames, which is relatively a small amount of time. Researchers such as Jing Wang use various methods like joint velocity to calculate blending length. In the present research transition duration/blending length is divided into six sections. Each section has approximately 2 to 10 frames to perform the transition. It is another good reason not to use any complex method to decide a simple 8-frame change. Instead, a simple effective method can well serve the purpose of transition.

Since eventually animation is the product to be experienced by users, users' (audiences') feedbacks are certainly relevant data to the development. For example, considering a motion transition of walking to running, despite root speed and other factors that can affect performance, there is a suitable blending length that can be performed in this research. An audience was asked to observe sets of movie clips and then respond by filling in a questionnaire.

5. Implementation

5.1 Pre-processed data

The ideal data set should be clearly defined and parameterised. The existing captured data is not qualified. The design described in chapter 4 is to convert a periodical data to a full cycle of this motion. The following code demonstrates how to accomplish that.

```

...

double diff[PM_MAX_FRAMES];

//the start point of full cycle technically can be any point.
//tmp = PostureDiff(mStartPosture, pMotionSample-
>m_pPostures[START+1]);
double MinimumDiff;

for(int i=START; i<mSampleFrameNo-1; i++)
{
    tmp = PostureDiff(mStartPosture, pMotionSample-
>m_pPostures[i+1]);
    diff[i] = CalTotalDiff(tmp);
    //debug
    //cout<< i<<" "<<diff[i] <<endl;

    //get the minimum difference frame
    //the differences of the frames next to START are the
    smallest but
    //we don't want them. Here hard coded SKIPFRAMES to
    eliminate this possibility

    if (i==START+SKIPFRAMES)
    {MinimumDiff = diff[i];
    }

    if (i>START+SKIPFRAMES)
    {
        if (diff[i]<MinimumDiff)
        {
            MinimumDiff = diff[i];
            endCycleFrame = i;
            cout<<i<<" "<<diff[i]<<endl;
        }
    }
}

```

...
 The *MinimumDiff* is the closest posture to the start posture which possibly is the end posture. The possibilities for *MinimumDiff* of getting small values especially in the periodical motion are high. It means that there are chances that in the middle of a full cycle motion a very “similar” posture (in value) to the start posture could be exist. The method of eliminating such possibility is to add a set of frames (*SKIPFRAMES*). Since initialising motion sequence is not part of automatic process is can be manually manipulated by the developer. The full cycle motion can be generated as following code (see Appendix Code *InitMotionData.cpp*).

```
...
void InitMotionData::regenerateMotion(int mScale, char* filename)
{
    //define a new motion
    Motion *cycleM = new Motion(filename,
    MOCAP_SCALE, pMotionSample->pActor);
    Motion *fullM = new Motion((endCycleFrame-
    START)*mScale, pMotionSample->pActor);

    for(int i=0; i<mScale; i++)
    {
        for(int j=0; j<cycleM->m_NumFrames; j++)
        {
            fullM->m_pPostures[i*cycleM->m_NumFrames+j] =
            cycleM->m_pPostures[j];
        }
    }
    //modify root
    int ei = cycleM->m_NumFrames-1;

    fullM;

    //get the length of the cycle
    double cyclePositionLth = cycleM-
    >m_pPostures[ei].root_pos.p[2] - cycleM-
    >m_pPostures[0].root_pos.p[2];
    cout<<endl<<"Cycle Length: "<<cyclePositionLth<<endl;

    for(i=1; i<mScale; i++)
    {
        for(int j=0; j<cycleM->m_NumFrames; j++)
        {
```

```

        fullM->m_pPostures[i*cycleM-
>m_NumFrames+j].root_pos.p[2] = cycleM-
>m_pPostures[j].root_pos.p[2]+i*cyclePositionLth;
    }
}

// generate the new motion with new file name "full"
string s1;
s1 = "full";
string s2(filename);
char* cycleFileName= (char*)s2.insert(0,s1).c_str();

//write the .amc file
fullM->writeAMCfile(cycleFileName,MOCAP_SCALE);
}

```

5.2 Blending and Warping

LERP is executed through a method called `LinearInterpolate()`. See Appendix Code `posture.cpp`.

```

Posture LinearInterpolate(float t, Posture const& a, Posture
const& b )
{
    Posture InterpPosture;

    //Interpolate root position
    InterpPosture.root_pos = interpolate(t, a.root_pos,
b.root_pos);

    //Interpolate bones rotations
    for (int i = 0; i < MAX_BONES_IN_ASF_FILE; i++)
    {
        InterpPosture.bone_rotation[i] = interpolate(t,
a.bone_rotation[i], b.bone_rotation[i]);
        InterpPosture.bone_translation[i] = interpolate(t,
a.bone_translation[i], b.bone_translation[i]);
    }
    return InterpPosture;
}

```

Inside the above method the attributes of posture object was further interpolated by another method `interpolate()` in vector level. See Appendix Code `vector.cpp`

```
vector interpolate( float t, vector const& a, vector const& b )
{
    return a*(1.0-t) + b*t;
}
```

Key frames/postures are defined in design section. The following code (see Appendix Code TwoMotion.cpp) describes the process.

```
...
Posture M1,AV1,AV2,M21,M22;
//calculate Postions inbetween first;
Posture positionPost[60];

//set the time cost for each frames float fInterpD =
1.0/(gap + 1.0);

for(int j=1;j<=gap;j++)
{
    positionPost[j] = LinearInterpolate(fInterpD*j,
pMotionOne->m_pPostures[nNumFramesInMotion1-1],
pMotionTwo->m_pPostures[0]);
}
//fill the new motion with first motion
pMotionZero->SetPosture(0, pMotionOne->m_pPostures[0]);

for(int i=1;i<nNumFrames;i++)
{
    int positionIndex = i-nNumFramesInMotion1+1;
//from 0 to end of motion one
if(i<nNumFramesInMotion1)
{
    pMotionZero->SetPosture(nCurPostureIndx,
pMotionOne->m_pPostures[i]);
    nCurPostureIndx++;
}
//MotionOne to Posture M1
else if(i==nNumFramesInMotion1 )
{
    float fInterpDist = 1.0/(gap + 1.0);
    for(int j=1;j<=10;j++)
    {
        float fTemp = fInterpDist*j;
        M1 = FetchPost(pMotionOne, 2);
        Posture InterPost =
LinearInterpolate(fInterpDist*j,
pMotionOne-
>m_pPostures[nNumFramesInMotion1 -1],M1);
    }
}
}
```

```

        InterPost.root_pos =
positionPost [positionIndex].root_pos;
        pMotionZero->SetPosture (nCurPostureIndx,
InterPost);
        nCurPostureIndx++;
    }
}

//Posture M1 to posture AV1
...
//Posture AV1 to posture AV2
...

```

The above code creates a new motion object *pMotionZero*. The key posture in-between two motion sequences are declared. The time cost of each frame is calculated. The new motion was filled with the first motion. In the section from the end of the first motion to the first key posture M1, method *LinearInterpolate()* does the LERP with newly defined M1 and other parameters. The other parts of the in-between of two motions follow the same procedural.

5.3 Catmull-Rom Application

Linear interpolation can cause sharp edge at the position of keyframe. It makes the movement of the character look like a bit stiff. In order to smooth movement and not to change the characteristics of the motion Catmull-Rom spline was chosen for the reason of its smoothing the motion at low computational cost. Motion data which includes joint rotations and root translations are both implemented using the Catmull-Rom Spline method. In 3D space the Catmull-Rom splines are defined almost the same as in 2D space. The difference is that in 2D space control points

are 2D points with coordinate(x, y), while in 3D space control points are vectors with coordinate (x, y, z).

Since motion data is linearly interpolated and warped there should not be any extra frames created and added into motion. Parameter t should be chosen by considering blend length and the length of each section, which is one sixth of blend length. In the Equation 2-21, the first control point of spline is not well defined. But it does not affect result of smooth the major section of the motion which is in the middle section of the motion. For this reason, the first and last frames are left untouched and the rest are manipulated through Catmull-Rom spline.

The Catmull-Rom spline was used to smooth the motion. The following code demonstrates the process. see Appendix Code vector.cpp

```

...
vector CatmullRom(const vector v1,const vector v2,const vector
v3,const vector v4,float t)
{
float t2 = t * t;
float t3 = t2 * t;
vector out ;

out.p[0]= 0.5f * ( ( 2.0f * v2.p[0] ) +( -v1.p[0] + v3.p[0] ) * t
+( 2.0f * v1.p[0] - 5.0f * v2.p[0] + 4 * v3.p[0] - v4.p[0] ) * t2
+( -v1.p[0] + 3.0f * v2.p[0] - 3.0f * v3.p[0] + v4.p[0] ) * t3 );
out.p[1]= 0.5f * ( ( 2.0f * v2.p[1] ) +( -v1.p[1] + v3.p[1] ) * t
+( 2.0f * v1.p[1] - 5.0f * v2.p[1] + 4 * v3.p[1] - v4.p[1] ) * t2
+( -v1.p[1] + 3.0f * v2.p[1] - 3.0f * v3.p[1] + v4.p[1] ) * t3 );
out.p[2]= 0.5f * ( ( 2.0f * v2.p[2] ) +( -v1.p[2] + v3.p[2] ) * t
+( 2.0f * v1.p[2] - 5.0f * v2.p[2] + 4 * v3.p[2] - v4.p[2] ) * t2
+( -v1.p[2] + 3.0f * v2.p[2] - 3.0f * v3.p[2] + v4.p[2] ) * t3 );

return out;
}

```

...

The above code is slightly difficult to read. The reason of not using matrix to tidy it up is because this is less expensive. When this method is executed in real-time the advantage can be significant. The following code shows the above method is used in a posture object. See Appendix Code posture.cpp

```

Posture CatmullRomInterpolate( Posture const& p1, Posture const&
p2, Posture const& p3, Posture const& p4, float t )
{
    Posture InterpPosture;

    //Interpolate root position
    InterpPosture.root_pos = CatmullRom( p1.root_pos,
p2.root_pos, p3.root_pos, p4.root_pos, t );

    //Interpolate bones rotations
    for (int i = 0; i < MAX_BONES_IN_ASF_FILE; i++)
    {
        InterpPosture.bone_rotation[i] =
CatmullRom(p1.bone_rotation[i],
p2.bone_rotation[i], p3.bone_rotation[i], p4.bone_rotation[i], t);
        InterpPosture.bone_translation[i] = CatmullRom(
p1.bone_translation[i],
p2.bone_translation[i], p3.bone_translation[i], p4.bone_translation[
i], t);
    }
    return InterpPosture;
}
...

```

The result of applying Catmull-Rom spline is significant. The stiff movements have been smoothed to generate natural looking movement, see Figure 5-1 The Catmull-Rom spline solved the problem of applying fixed section.

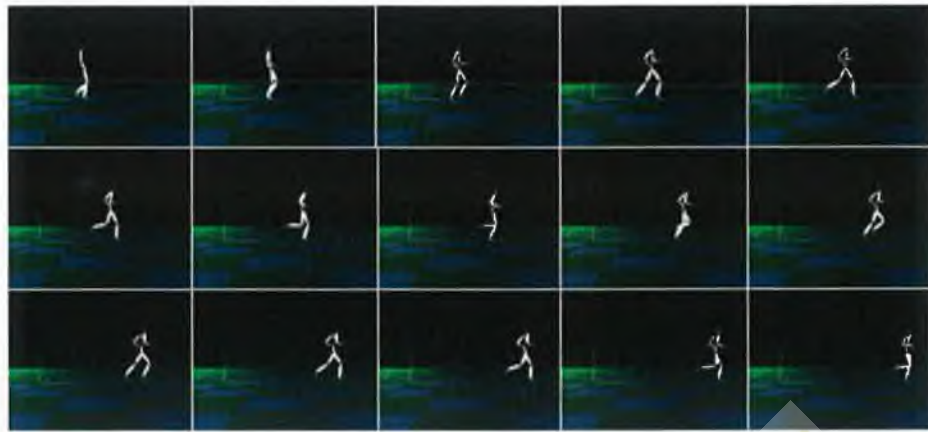


Figure 5-1

lyit | Institiúid Teicneolaíochta Leitir Ceannlann
Letterkenny Institute of Technology

6. Evaluation

6.1 Survey Objective

The objective of this survey is to find out the relationship between blending length and the quality of the transition. The expected result is that in certain length period the transition should be smooth and realistic. It is expected that the majority amount of respondents chose one of the clips as the best. If the result is not significant e.g. respondents vote evenly to each movie clip, then the blending length should be replaced by a wider range i.e. if the set of movie clips with blending length 12, 18 and 24 frames the result shows no clear winner, then a wider range of blending length 6, 18 and 30 frames should be used.

6.2 Questionnaire Design

The purpose of the questionnaire is to test differences between movie clips with various blending lengths and to pick the best blending length which makes the transition look most realistic.

In order to make the survey more objective and effective, there are some issues that should be addressed. Firstly, the rating should be straightforward because it would not be obvious what were the second best and the third best. So three ratings – best, middle and worst- are given.

Secondly, the audiences might consciously choose the particular one if blending length of clips in a set are presented in a continuous increasing order. Although the audience members do not know the exact length of frames they can feel that the blending length is increasing. This fact might affect their judgement. The random order is given when clips are presented and the real blending length is unknown to participants.

Thirdly, the choice of audience can be young and familiar with video presentation because they are the major potential target of the product. Audiences chosen in this research mostly are third year Computer Game Development students in Letterkenny Institute of Technology. These participants are in the area of computer animation and have knowledge of 3D graphics. In other words, the participants knew what to look at and they provided their, considered opinions.

Some other issues like the choice of blending length and presentation also need to be considered. See Appendix 6.1 survey sheet used in this research.

6.3 Results and Analysis

The following Table 4-1 shows the result of the survey.

Run to Walk (1 shortest 2 middle 3 longest)																	
best	2	2	2	2	3	3	3	3	3	3	2	3	1	3	3	1	
worst	1	3	1	1	1	1	1	1	1	1	1	3	1	3	1	1	3
Walk to Run (1 middle 2 shortest 3 longest)																	
best	1	1	1	1	2	3	2	1	2	1	3	1	2	1	1	1	3
worst	2	2	3	2	3	2	3	3	3	3	2	2	3	2	3	3	2
Walk to Jump (1 longest 2 shortest 3 middle)																	
Best	1	1	1	1	3	1	1	1	1	1	2	2	1	3	1	1	3
Worst	3	3	3	3	2	2	3	2	3	2	3	3	3	1	3	3	1

	count		percentage				total
Run to Walk	longest	%	middle	%	shortest	%	
	5	29.4	10	58.8	2	11.8	17
	4	23.5	0	0	13	76.5	17
Walk to Run							
	3	17.6	10	58.8	4	23.5	17
	9	52.9	0	0	8	47.1	17
Walk to Jump							
	12	70.6	3	17.6	2	11.8	17
	2	11.8	11	64.7	4	23.5	17

Table 6-1 Survey Result of Transition Period

In this survey the blending length settings are longest (30 frames), middle (24 frames) and shortest (18 frames). The respondents only need to rate the best quality and the worst one.

58.8% respondents think the best quality clip from "run to walk" is with the middle blending length. 58.8% people think the best quality clip from "walk to run" is with the middle blending length. 70.6% people think the best quality clip from "walk to jump" is with the longest blending length. A clear majority suggested that the clip with longest blending length was the best.

Result of "run to walk" shows the best blending length is the middle length clip (24 frames). And the worst clip is the one with shortest blending length, though it is not the general case that the shortest blending length is always worse than the longer one. In contrast, Table 4-1 shows that with 52.9% clip "walk to run" has the worst result with the longest blending length. The relationship between blending length and motion speed or other factors can be explored in further study.

The surprising result is the green 64.7% because if the blending length is the factor that makes a big difference then the clip with shortest blending length should be the worst since the clip with the longest blending length is the best one. But have the original motion and target motion are very dissimilar, and the result is informative, though a bit surprising.

6.4 Comments and Issues to be Concerned

Along with the survey questionnaire comments on the animation performance as well as opinions on the projects were requested from respondents.

One respondent writes that “in the run to walk motion the transition looks too much like the character is sliding.” This is a standard artefact of such transitions. As introduced earlier in this paper foot sliding can be prevented by implementing inverse kinematics. However, implementing inverse kinematics to animation system is a complex process. The limitation of time makes IK difficult to apply. In addition, IK is not the focus of this research.

Another respondent writes that “The skeleton seems to slightly rise when the animation changes.” In the transition from walk to jump the model seems to slide along Z-plane a little before jump.” By observing motion transition carefully it is the fact that there are some artefacts that can be detected like the above respondents noted. Animation characters have slight displacement. In this case the skeleton rises slightly because of root position in Y axes. In the research, root positions are calculated based on the original motion speed and target motion speed. In order to concentrate on speed changes, a number of complex situations were avoided when dealing with root position, such as the character attitude problem when path diagram intersects itself. Root positions are manipulated for the presentation purpose in this research.

Another respondent writes that “maybe another half step run might make it look better.” When the character stops, I think it might better if it planted to a stop”. The choice of which foot and when to start transition is an important issue and key to the success of transition. In fact a method can be developed by using certain types of motions, target foot position and speed and so on. In this research no inputs for influencing the animation are expected in real-time, so transition method should consider all kinds of problems. Due to time limitation these problems have not been addressed in this work, but might be part of further research in this area.

6.5 Conclusion

The survey conducted broadly supports Wang and Bodenheimer’s work (Wang 2004) suggesting that the time duration of a transition is of critical importance to how the physical transition is perceived.

In the case of the transition from a walk to a jump there was an indication that a slightly shorter transition time was favoured. It is felt that this may be mainly due to the fact the motions are highly dissimilar.

In the next chapter conclusions and further work are considered.

7. Conclusions and Further Work

7.1 Conclusion

Early in this thesis, questions have been posed regarding the problem of motion blending. At this point it is important to consider to what extent they have been answered.

Can automated blending of similar motions, such as walk and run be achieved at interactive rates, thereby increasing the utility of a database of motion without user input?

This question can be partially answered. Despite the condition of interactive rates, the answer to this question is “yes, it can”. The reason why an interactive system was not used is that such system would have been very complex and time-consuming to develop. In chapter 2, achievements of other researches are reviewed and a simple structured method suiting a real-time system is determined. In chapter 4, the process of constructing such a method is explained in detail. In this method both characteristics of existing motion sequences are chosen and mixed together through blending/warping. The root positions are calculated based on velocities and accelerations of the motion. Artefacts could be removed through different methods and techniques. For example, foot sliding can be removed via

implementing IK. Stiffness of body movements are removed by executing Catmull-Rom splines. The sample results are smooth and realistic.

Can current work on blending/warping be extended to deal with a wider range of motion types?

This question can be answered with a qualified - "Yes, it can". The method is designed for similar motion sequences. However, dissimilar motion sequences are evaluated as well in the research. Motion 'walk' and 'jump up' are used. These two motions are very different, particularly in root speed and position. The evaluation result is acceptable; however, it could be improved by introducing additional methods for dealing with dissimilar motion sequences. One of the methods would be to define an intermediate pose between two motions, such as a standing position. Further study in this area would be very useful.

Can artefacts of blending/warping such as foot skate or self-intersection be eliminated at interactive speed?

The answer is yes but it has not been tested at interactive speed. Artefacts of stiff body movement are removed through applying Catmull-Rom spline. The result is significant. When character moves from walk to run it shows a very smooth hand swing, foot swing and no sudden movements. Foot skating can be removed via IK.

All these questions have been answered or partially answered. The remaining parts are left unanswered for future work.

7.2 Future work

The research chose to divide the transition into a number of sections and interpolate between the feature of the start motion and target motion by choosing a fixed frame in both motion data. This operation can result in an unpredictable motion. Transitions between dissimilar motions are difficult but there are certain methods that could be developed in order to improve the transitions, such as introducing an intermediate pose. This area could definitely benefit from further study.

Certain areas went beyond the focus of this research or could not be pursued due to the time limitation.

Early in the development of this project a quaternion class was developed with a view that it might be utilised for producing smooth interpolations. The methods implemented in this work did not require the power of quaternion interpolation, but it is felt that smoother, more realistic, and computationally cheap transitions/warps might be achieved by using such methods.

Another question that rose in the development was how to find, automatically, optimal frames in motion sequences being joined so that the transition will occur

most naturally, both physically and visually. This was considered a difficult problem, involving such questions as identifying when a foot was on the “ground”, as well as computing the relative energies of the motions further research would attempt to deal with these issues to produce enhanced transitions at low cost.

Available memory and increased processor speeds are likely to result in making methods, which previously were only feasible to do “off-line”, into real-time methods. In particular a development will most likely effect the methods available for real-time transitions/warps is that of the graphics processor unit (GPU). Already the GPU is, in many cases, been used for jobs other than strictly graphical work.

Another possible development in hardware that might bring transition/warp methods into the real-time domain is if quaternion arithmetic were available in hardware.

Whatever developments occur it seems reasonable to expect that the intense interest in, and impressive developments of the whole area of computer animations will continue and probably accelerate.

References

- Amaya, K., Bruderlin, A., and Calvert, T. (1996). "Emotion from motion." Graphic Interface (W.A. Davis and R.Bartels.): 222-229.
- Arikan, O., Forsyth, D.A.&O'Brein, J.F. (2003). "Motion synthesis from annotations." ACM Transactions on Graphics **22** 21(3): 465-472.
- Arikan, O. F., D.A. (2002). "Interactive motion generation from examples." ACM Transactions on Graphics(Proceedings of SIGGRAPH'02) **21**(3): 483-490.
- Babbie, E. (2001). The Practice of Social Research. Belmont California, Wadsworth Publishing Company.
- Ball, R. (2008). "Oldest Animation Discovered In Iran." Animation Magazine.
- Bruderlin, A., and Williams, L. (1995). "Motion signal processing." Computer Graphics(proceedings of SIGGRAPH 85): 263-270.
- Catmull, E. R., R. (1974). A Class of Local Interpolating Splines. NY.
- Cooper, D. R. E., C.W. (1998). Business research methods. irwin, Chicago.
- Di Fiore, F., Schaeken, P., Elens, K., & Van Reeth, F. (2001). "Automatic In-betweening in Computer Assisted Animation by Exploiting 2.5D Modelling Techniques." Proceedings of Computer Animation 2001: 192–200.
- Foley, J. D. (1997). Computer graphics: Principles and Practice, Addison-Wesley.
- Guo, S. R., J. (1996). "A high-level control mechanism for human locomotion based on parametric frame space interpolation." Eurographics Workshop on Computer Animation and Simulation 96: 95-107.
- Hamilton, W. R. (1844). "On a new species of imaginary quantities connected with a theory of quaternions." Royal Irish Academy **2**: 424-434.
- Kovar, L. G., M. (2002). "Motion Graph." ACM Transactions on Graphics(Proceedings of SIGGRAPH'02) **21**(3): 473-481.
- Kovar, L. G., M. (2003). "Flexible automatic motion blending with registration curves." Symposium on Computer Animation: 214-224.
- Kovar, L. G., M. (2004). "Automated extraction and parameterization of motions in large data sets " ACM Transactions on Graphics (TOG) **23**(3).

References

- Lee, J., Chai, J., Reitsma, P.S.A., Hodgins, J.K. & Pollard, N.S. (2002). "Interactive control of avatars animated with human motion data." ACM Transactions on Graphics **21**(3): 491-500.
- Les, A. P. W., T. (1997). The NURBS Book, Springer.
- Menache, A. (1999). Understanding Motion Capture for Computer Animation and Video Games Morgan Kaufmann.
- Mizuguchim, B. J. C. T. (2001). "Data driven motion transitions for interactive games." Eurographics 2001 Short Presentations.
- Nancy S. Pollard, J. K. H., Marcia J. Riley, Christopher G. Atkeson (2002). "Adapting Human Motion for the Control of a Humanoid Robot." Robotics and Automation, 2002. Proceedings. ICRA '02. IEEE International Conference **2**: 1390-1397.
- Perlin, K. (1995). "Real time responsive animation with personality." IEEE Transactions on Visualization and Computer Graphics **1**: 5-15.
- Pullen, K. B., C. (2000). "Motion capture assisted animation: Texturing and synthesis." In Proceedings of ACM SIGGRAPH 2002. Annual Conference Series, ACM SIGGRAPH.
- Qiang, Y., Geoff, W. & Geoffrey, I. W. (2006). PRICAI 2006: Trends in Artificial Intelligence Springer.
- Reitsma, P. S. A. P., N. S. (2007). "Evaluating motion graphs for character animation." ACM Transactions on Graphics (TOG) **26**(4): 18.
- Rose, C., Cohen, M.F. & Bodeheimer, B. (1998). "Verbs and Adverbs." IEEE Computer Graphics and Applications **18**(5): 32-40.
- Rose, C., Guderter, B., Bodeheimer, B. & Cohen, M. F. (1996). "Efficient generation of motion transitions using space time constraints." ACM press: 147-154.
- Rose, C., Sloan, P. & Cohen, M.F. (2001). "Artist-directed inverse kinematics using radial basis function interpolation." Proceedings of Eurographics 2001 **20**(3).
- Sang, I. P., Hyun, J. S., Tae, H. K. & Sung, Y. S. (2004). "On-line motion blending for real-time locomotion generation." Comp. Anim. Virtual Worlds 2004 **15**: 125-138.
-

References

- Saunders, L. T. (2003). Research Methods for Business Studies, Prentice Hall / Financial Times.
- Sidenbladh, H., Black, M. J. & Signal (2002). "Implicit probabilistic models of human motion for synthesis and tracking." Computer Vision ECCV 2002 (1): 784-800.
- Sloan, P., Rose, C.F. & Cohen, M.F. (2001). "Shape by example." 2001 ACM Symposium on Interactive 3D Graphics: 135-144.
- Unuma, M., Anjyo, K., and Tekeuchi, R (1995). "Fourier principles for emotion-based human figure animation." Computer Graphics (proceedings of SIGGRAPH 95): 91-96.
- Wang, J. B., B. (2008). "Synthesis and evaluation of linear motion transitions." ACM Transactions on Graphics (TOG) 27(1).
- Wang, J. B., B. (2003). "An evaluation of a cost metric for selecting transitions between motion segments." Symposium on Computer Animation 232-238.
- Wang, J. B., B. (2004). "Computing the Duration of Motion Transitions: An Empirical Approach." Eurographics/ACM SIGGRAPH Symposium on Computer Animation 335 - 344.
- Wiley, D. J. H., J.K. (1997). "Interpolation synthesis for articulated figure motion." IEEE Computer Graphics and Applications 17(6): 39-45.
- Witkin, A. P., Z. (1995). "Motion warping." Computer Graphics (proceedings of SIGGRAPH 95): 105-108.

Appendix I - Motion Blending Length Survey Table

Motion Clip: Run To Walk			
Blending Length	Quality Rating (1. Poor 2. Medium 3. Good)		
18			
24			
36			

Motion Clip: Walk To Run			
Blending Length	Quality Rating (1. Poor 2. Medium 3. Good)		
18			
24			
36			

Motion Clip: Walk to Jump			
Blending Length	Quality Rating (1. Poor 2. Medium 3. Good)		
18			
24			
36			

Name:

Date:

Appendix II - Code

lyit | Institiúid Teicneolaíochta Leitir Ceanaínn
Letterkenny Institute of Technology

```
//-----  
// FileName: posture.h  
// Description: this is the header file  
// of posture class which defines postures  
// and methods used to manipulate the motion.  
//-----  
  
#ifndef _POSTURE_H  
#define _POSTURE_H  
  
#include "vector.h"  
#include "types.h"  
  
//Root position and all bone rotation angles (including root)  
class Posture  
{  
    public:  
        //linear interpolation  
        friend Posture LinearInterpolate(float, Posture const&, Posture const& );  
  
        //catmull-rom method for smoothing the motion  
        friend Posture CatmullRomInterpolate(Posture const&, Posture const&, Posture const&, Posture  
const&, float );  
  
        //calculate the differences of joint angles all over the body between  
        friend Posture PostureDiff(Posture const&, Posture const& );  
        friend double CalTotalDiff(Posture const&);  
  
        //method for letting skeleton walk along a straight line  
        friend Posture StraightLine(Posture const& a, Posture const& b, Posture &k, int steps);  
        friend Posture GoStraight(Posture const& a, Posture const& b);  
  
        //methods for manipulating root position  
        friend double DistanceZ(Posture const& a, Posture const& b);  
        friend void moveToBaseZ(double dis, Posture & b );  
        friend double getRootZ(Posture a);  
        friend double DistanceX(Posture const& a, Posture const& b);  
        friend void moveToBaseX(double dis, Posture & b );  
};
```



```
friend double getRootX(Posture a);

//generate a new posture using linear interpolation
friend Posture AveragePostures(Posture const&, Posture const&, double scale);

//member variables
public:

//Root position (x, y, z)
vector root_pos;

//Rotation (x, y, z) of all bones at a particular time frame in their local coordinate system.
//If a particular bone does not have a certain degree of freedom,
//the corresponding rotation is set to 0.
//The order of the bones in the array corresponds to their ids in .ASf file: root, lhipjoint, lfemur, ...
vector bone_rotation[MAX_BONES_IN_ASF_FILE];
vector bone_translation[MAX_BONES_IN_ASF_FILE];
vector bone_length[MAX_BONES_IN_ASF_FILE];
};

#endif
```

```
//-----  
// FileName: posture.cpp  
// Description: this is the cpp file  
// of posture class which defines postures  
// and methods used to manipulate the motion.  
// processed data can be stored in a database.  
//-----  
  
#include "posture.h"  
  
//Output Posture = (1-t)*a + t*b  
Posture LinearInterpolate(float t, Posture const& a, Posture const& b )  
{  
    Posture InterpPosture;  
  
    //Interpolate root position  
    InterpPosture.root_pos = interpolate(t, a.root_pos, b.root_pos);  
  
    //Interpolate bones rotations  
    for (int i = 0; i < MAX_BONES_IN_ASF_FILE; i++)  
    {  
        InterpPosture.bone_rotation[i] = interpolate(t, a.bone_rotation[i], b.bone_rotation[i]);  
        InterpPosture.bone_translation[i] = interpolate(t, a.bone_translation[i], b.bone_translation[i]);  
    }  
    return InterpPosture;  
}  
  
Posture CatmullRomInterpolate( Posture const& p1, Posture const& p2, Posture const& p3, Posture const& p4, float  
t )  
{  
    Posture InterpPosture;  
  
    //Interpolate root position  
    InterpPosture.root_pos = CatmullRom( p1.root_pos, p2.root_pos, p3.root_pos, p4.root_pos, t);  
  
    //Interpolate bones rotations  
    for (int i = 0; i < MAX_BONES_IN_ASF_FILE; i++)  
    {
```

```
        InterpPosture.bone_rotation[i] = CatmullRom(p1.bone_rotation[i],
p2.bone_rotation[i],p3.bone_rotation[i],p4.bone_rotation[i],t);
        InterpPosture.bone_translation[i] = CatmullRom( p1.bone_translation[i],
p2.bone_translation[i],p3.bone_translation[i],p4.bone_translation[i],t);
    }
    return InterpPosture;
}
```

```
//get the difference of joint angles and return the posture of difference
Posture PostureDiff(Posture const& P1 , Posture const& P2 )
```

```
{
    Posture result;
    for (int i = 0; i < MAX_BONES_IN_ASF_FILE; i++)
    {
        result.bone_rotation[i] = P1.bone_rotation[i] - P2.bone_rotation[i];
    }
    return result;
}
```

```
// add all the difference of all the bones together
double CalTotalDiff(Posture const& p)
```

```
{
    double total=0.0;
    for (int i = 0; i < MAX_BONES_IN_ASF_FILE; i++)
    {
        total += len(p.bone_rotation[i]);
    }
    return total;
}
```

```
Posture StraightLine(Posture const& a,Posture const& b,Posture &k,int steps)
```

```
{
    Posture ResultPos;

    //a straight line defined by  $v = v1+(v2-v1)t$ 
```

```
//calculate t
double t = Distance(a.root_pos,b.root_pos);

ResultPos.root_pos= a.root_pos+(b.root_pos - a.root_pos)*t*steps;
//to make the line paraell to ground
ResultPos.root_pos.setValue(1,k.root_pos.p[1]);

for (int i = 0; i < MAX_BONES_IN_ASF_FILE; i++){
    ResultPos.bone_rotation[i] = k.bone_rotation[i];
    ResultPos.bone_translation[i] = k.bone_translation[i];
}
return ResultPos;
}

//calculate the distance in the direction along z-axis
double DistanceZ(Posture const& a,Posture const& b )
{
    return b.root_pos.p[2]-a.root_pos.p[2];
}

//calculate the base of z-axis
void moveToBaseZ(double dis,Posture & b )
{
    b.root_pos[2] = b.root_pos.p[2] - dis;
}

//calculate the root position along z-direction
double getRootZ(Posture a)
{
    double iniZ = a.root_pos.p[2];
    return iniZ;
}

//calculate the distance in the direction along x-axis
double DistanceX(Posture const& a,Posture const& b )
{
```

```
    return b.root_pos.p[0]-a.root_pos.p[0];
}

//calculate the base of x-axis
void moveToBaseX(double dis,Posture & b )
{
    b.root_pos[0] = b.root_pos.p[0] - dis;
}

//calculate the root position along x-direction
double getRootX(Posture a)
{
    return a.root_pos.p[0];
}

//generate new postures by interpolating two motions
Posture AveragePostures(Posture const& a, Posture const& b, double scale)
{
    Posture InterpPosture;

    //Interpolate bones rotations
    for (int i = 0; i < MAX_BONES_IN_ASF_FILE; i++)
    {
        InterpPosture.bone_rotation[i] = a.bone_rotation[i]* scale + b.bone_rotation[i]*(1-scale);
        InterpPosture.bone_translation[i] = a.bone_translation[i]* scale+ b.bone_translation[i]*(1-scale);
    }
    return InterpPosture;
}
```

InitMotionData.h

```
//-----  
// FileName: InitMotionData.h  
// Description: this is the header file  
// of InitMotionData class which preprocesses  
// motion data for trnsition purpose. The  
// processed data can be stored in a database.  
//-----  
  
#ifndef _INITMOTIONDATA_H  
#define _INITMOTIONDATA_H  
  
#include "motion.h"  
  
class InitMotionData  
{  
public:  
    // initialize the motion captured data sequence  
    InitMotionData(Motion* pMotion);  
  
    // get a full cycle of one type of motion  
    int GetFullCycle();  
  
    // the method defines the start point  
    int GetStartPosture();  
  
    // regenerate a  
    void regenerateMotion(int mScale,char* filename);  
  
    //dataset  
    int startCycleFrame;  
    int endCycleFrame;  
    Motion * pMotionSample;  
  
    //restores a full cycle of a motion  
    Motion * pMotionCycle;  
};  
  
#endif
```

lyit

Institiúid Teicneolaíochta Lettir Ceannainn
Letterkenny Institute of Technology

```
//-----  
// FileName: InitMotionData.cpp  
// Description: this is the cpp file  
// of InitMotionData class which preprocesses  
// motion data for transition purpose. The  
// processed data can be stored in a database.  
//-----  
  
#include "InitMotionData.h"  
#include "posture.h"  
#include "vector.h"  
#include "types.h"  
  
#include <iostream>  
using namespace std;  
  
InitMotionData::InitMotionData(Motion * pMotion)  
{  
    endCycleFrame=0;  
    startCycleFrame=0;  
    pMotionSample=pMotion;  
}  
  
int InitMotionData::GetFullCycle()  
{  
    int mCycleFrames= 0;  
    int mSampleFrameNo = pMotionSample->m_NumFrames;  
    Posture mStartPosture;  
    mStartPosture = pMotionSample->m_pPostures[START];  
  
    Posture tmp;  
    double diff[PM_MAX_FRAMES];  
  
    //take the fist  
    //tmp = PostureDiff(mStartPosture,pMotionSample->m_pPostures[START+1]);  
    double MinumDiff;
```



```
for(int i=START;i<mSampleFrameNo-1;i++)
{
    tmp = PostureDiff(mStartPosture,pMotionSample->m_pPostures[i+1]);
    diff[i] = CalTotalDiff(tmp);
    //debug
    //cout<< i<<" "<<diff[i] <<endl;

    //get the minimum difference frame
    //the differences of the frames next to START are the smallest but
    //we don't want them. Here hard coded SKIPFRAMES to eliminate this possibility

    if (i==START+SKIPFRAMES)
    {MinimumDiff = diff[i];
    }

    if (i>START+SKIPFRAMES)
    {
        if (diff[i]<MinimumDiff)
        {
            MinimumDiff = diff[i];
            endCycleFrame = i;
            cout<<i<<" "<<diff[i]<<endl;
        }
    }
}
if (endCycleFrame<APPROX_CYCLE_NO)
    endCycleFrame *= 2;

//endCycleFrame = START + (endCycleFrame-START)/((endCycleFrame-START)/APPROX_CYCLE_NO+1);
cout<<START <<" "<< endCycleFrame;

return endCycleFrame;
}

int InitMotionData::GetStartPosture()
{
    //define the start posture by parameters
```

```
//there are many ways to define the start point
//the method can be developed in the future
return startCycleFrame;
}

void InitMotionData::regenerateMotion(int mScale,char* filename)
{
    //define a new motion
    Motion *cycleM = new Motion(filename, MOCAP_SCALE,pMotionSample->pActor);
    Motion *fullM = new Motion((endCycleFrame-START)*mScale,pMotionSample->pActor);

    for(int i=0;i<mScale;i++)
    {
        for(int j=0;j<cycleM->m_NumFrames;j++)
        {
            fullM->m_pPostures[i*cycleM->m_NumFrames+j] = cycleM->m_pPostures[j];
        }
    }

    //modify root
    int ei = cycleM->m_NumFrames-1;

    fullM;

    //get the length of the cycle
    double cyclePositionLth = cycleM->m_pPostures[ei].root_pos.p[2] - cycleM->m_pPostures[0].root_pos.p[2];
    cout<<endl<<"Cycle Length: "<<cyclePositionLth<<endl;

    for(i=1;i<mScale;i++)
    {
        for(int j=0;j<cycleM->m_NumFrames;j++)
        {
            fullM->m_pPostures[i*cycleM->m_NumFrames+j].root_pos.p[2] =
            cycleM->m_pPostures[j].root_pos.p[2]+i*cyclePositionLth;
        }
    }
}
```

```
// generate the new motion with new file name "full"
string s1;
s1 = "full";
string s2(filename);
char* cycleFileName= (char*)s2.insert(0,s1).c_str();

//write the .amc file
fullM->writeAMCfile(cycleFileName,MOCAP_SCALE);
```

lyit | **Institiúid Teicneolaíochta Lettir Ceannainn**
Letterkenny Institute of Technology

```
//-----  
// FileName: TwoMotions.h  
// Description: this is the header file  
// of TwoMotions class which defines trnsitions  
// between two motion sequences and manipulates  
// root position of new generated motion.  
//-----  
  
#ifndef _TWMOTIONS_H  
#define _TWMOTIONS_H  
  
#include "motion.h"  
#include "posture.h"  
  
class TwoMotions  
{  
public:  
  
    TwoMotions(Motion* pMotion1, Motion* Motion2);  
    TwoMotions(Motion* pMotion1, Motion* Motion2, char* pOffsetFileName);  
  
    //read frame offset files which can specify the frames of skipping  
    void ReadOffsetFile(char* pOffsetFileName);  
  
    //Interpolation with parameter gap  
    Motion * InterpolateTwoMotions(int gap);  
  
    //applying catmull-rom splines  
    void CatmullRomSpline(Motion *&);  
  
    //general linear interpolation  
    Motion * LinearInterp(int length);  
  
    //method for motions with middle frames with parameter-blending length  
    Motion * InterpSetMiddleFrames(int length);  
  
    //method for motions with middle frames
```

```
Motion * InterpSetMiddleFrames();

//two motions move along a straight line
Motion* GoStraightLine();

//two motions move to the same line
void MoveToSameLine();

//set empty frames between two motion sequences
Motion* SetGap(int gapLength);

//generate time offset files
void generateTimeOffSetFile();

Posture FetchPost(Motion* pMotion,int fNo);
Posture AveragePost(Posture &, Posture &,double);

//calculate velocity
double getVelocity(Motion* pMotion, double mVelocity);
double CalVelocity(Posture p1,Posture p2);

//calculate root velocity
vector CalRootVelolity(Posture p1,Posture p2,double frameRate);
vector RootVelocity(vector rootStart,vector rootEnd,double t);

//get unit vector
vector CalUnitVector(vector v);
vector AlphaT(vector v1,vector v2, double t);

//calculate root position
vector CalRootPostion(vector p1, vector vel1,vector vel2,double time);

double getV1();
double getV2();
```

```
void setV1(double v1);
void setV2(double v2);

Motion* getMotion1();
Motion* getMotion2();

private:

Motion * pMotionZero;
Motion * pMotionOne;
Motion * pMotionTwo;

double mVelocity1;
double mVelocity2;

int* m_pTimeDistArray;
int blendLength;

InterpType m_InterpTypeToUse;

//Angle representation (euler angles and quaternians)
AngleRepresent m_AngleRepresToUse;
ErrorType m_ErrorType;

void LinearInterpEulerAngles_TwoMotions(Motion* pInterpMotion1, Motion* pInterpMotion2);
};
#endif
```

```
//-----  
// FileName: TwoMotions.cpp  
// Description: this is the cpp file  
// of TwoMotions class which defines trnsitions  
// between two motion sequences and manipulates  
// root position of new generated motion.  
//-----  
  
#include "twoMotions.h"  
#include <iostream>  
using namespace std;  
  
// constructor with initial two motion sequences  
TwoMotions::TwoMotions(Motion * pMotion1, Motion * pMotion2)  
{  
    //pMotionOne is start motion; pMotionTwo is target motion  
    pMotionOne = pMotion1 ;  
    pMotionTwo = pMotion2 ;  
  
    //pMotionZero is the new motion will be created  
    pMotionZero = NULL;  
  
    //two motion's root velocities  
    mVelocity1 = 0.0;  
    mVelocity2 = 0.0;  
  
    //Set default interpolation type  
    m_InterpTypeToUse = LINEAR;  
  
    //set default angle representation to use for interpolation  
    m_AngleRepresToUse = EULER;  
  
    //Set ErrorType to NO_ERROR  
    m_ErrorType = NO_ERROR_SET;  
  
    //Init m_pTimeDistArray array  
    m_pTimeDistArray = NULL;  
}
```



```
//set default blendLength to zero
blendLength = 0;
}

// constructor with initial two motion sequences and offsetFile
// offsetfile is used for another method of warping the motion
// this method can let motion skips certain frames in between manually
// by inputting the frames number

TwoMotions::TwoMotions(Motion* pMotion1, Motion* pMotion2, char* pOffsetFileName)
{
    //pMotionOne is start motion; pMotionTwo is target motion
    pMotion1 = pMotionOne;
    pMotion2 = pMotionTwo;

    //two motion's root velocities
    mVelocity1 = 0.0;
    mVelocity2 = 0.0;

    //Set default interpolation type
    m_InterpTypeToUse = LINEAR;

    //set default angle representation to use for interpolation
    m_AngleRepresToUse = EULER;

    //Set ErrorType to NO_ERROR
    m_ErrorType = NO_ERROR_SET;

    //Init m_pTimeDistArray array
    m_pTimeDistArray = NULL;
    ReadOffsetFile(pOffsetFileName);
}

void TwoMotions::ReadOffsetFile(char* pOffsetFileName)
{
```

```
//open the file
FILE* pInFile = fopen(pOffsetFileName, "r");
//AD_OFFSET_FILE are defined in types.h
if (pInFile == NULL)
{
    m_ErrorType = BAD_OFFSET_FILE;
    return;
}

//Allocate memory for m_pTimeDistArray
m_pTimeDistArray = new int [pMotionZero->m_NumFrames];

int frameNum, frameNumPrev = 0;
//read the file
for (int i = 0; i < pMotionZero->m_NumFrames; i++)
{
    //read next line
    if (fscanf(pInFile, "%d", &frameNum) == -1)
    {
        m_ErrorType = BAD_OFFSET_FILE;
        return;
    }

    //Compute offset and store into array
    //offset = number of frames skipped between frame i and i-1
    m_pTimeDistArray[i] = frameNum - frameNumPrev - 1;
    frameNumPrev = frameNum;
}
}

Motion * TwoMotions::InterpolateTwoMotions(int gap)
{
    //Do nothing if error is set
    if (m_ErrorType != NO_ERROR_SET)
    {
        pMotionZero = NULL;
        // pMotionTwo = NULL;
    }
}
```

```
        return NULL;
    }
    //Compute number of frames in the new (interpolated) motion
    int nNumFrames = 0;
    int nNumFramesInMotion1 = pMotionOne->getNumOfFrames();
    int nNumFramesInMotion2 = pMotionTwo->getNumOfFrames();
    nNumFrames =nNumFramesInMotion1+nNumFramesInMotion2+gap;

    //create new motion - initially set to default motion
    pMotionZero= new Motion(nNumFrames,pMotionOne->getActor());

    //Perform the interpolation
    if (m_InterpTypeToUse == LINEAR && m_AngleRepresToUse == EULER) {
        LinearInterpEulerAngles_TwoMotions(pMotionOne,pMotionTwo);
        return pMotionZero;
    }
    else
    {
        //For now only linear interpolation of euler angles is supported
        m_ErrorType = NOT_SUPPORTED_INTERP_TYPE;
        delete pMotionZero;
        pMotionZero = NULL;
        return NULL;
    }
}

//interpolate two motions using method of set middle frames (key frames)with parameter gap
Motion * TwoMotions::InterpSetMiddleFrames(int gap)
{
    //Do nothing if error is set

    if (m_ErrorType != NO_ERROR_SET)
    {
        pMotionZero = NULL;
        // pMotionTwo = NULL;
    }
}
```

```
    return NULL;
}
//Compute number of frames int the new (interpolated) motion
int nNumFrames = 0;
int nNumFramesInMotion1 = pMotionOne->getNumOfFrames();
int nNumFramesInMotion2 = pMotionTwo->getNumOfFrames();
nNumFrames =nNumFramesInMotion1+nNumFramesInMotion2+gap;

//Allocate new motion - initially set to default motion
pMotionZero= new Motion(nNumFrames,pMotionOne->getActor());

int nCurPostureIndx = 1;
Posture M1,AV1,AV2,M21,M22;

//calculate Postions inbetween firt;
Posture positionPost[60];

//set the time cost for each frames
float fInterpD = 1.0/(gap + 1.0);
//
for(int j=1;j<=gap;j++)
{
    positionPost[j] = LinearInterpolate(fInterpD*j, pMotionOne->m_pPostures[nNumFramesInMotion1-1],
        pMotionTwo->m_pPostures[0]);
}

pMotionZero->SetPosture(0, pMotionOne->m_pPostures[0]);

for(int i=1;i<nNumFrames;i++ )
{
    int positionIndex = i-nNumFramesInMotion1+1;
    //from 0 to end of motion one
    if(i<nNumFramesInMotion1)
    {
        pMotionZero->SetPosture(nCurPostureIndx, pMotionOne->m_pPostures[i]);
        nCurPostureIndx++;
    }
}
```

```
//MotionOne -m1
else if(i==nNumFramesInMotion1 )
{
    float fInterpDist = 1.0/(gap + 1.0);
    for(int j=1;j<=10;j++)
    {
        float fTemp = fInterpDist*j;
        M1 = FetchPost(pMotionOne, 2);
        Posture InterPost = LinearInterpolate(fInterpDist*j,
            pMotionOne->m_pPostures[nNumFramesInMotion1 -1],M1);
        InterPost.root_pos = positionPost[positionIndex].root_pos;
        pMotionZero->SetPosture(nCurPostureIndx, InterPost);
        nCurPostureIndx++;
    }
}

// av1
else if(i==nNumFramesInMotion1+10 )
{
    float fInterpDist = 1.0/(gap + 1.0);
    for(int j=1;j<=10;j++)
    {
        float fTemp = fInterpDist*j;
        Posture tmpPost1 = FetchPost(pMotionOne, 4);
        Posture tmpPost2 = FetchPost(pMotionTwo, nNumFramesInMotion2-4);
        AV1 = AveragePost(tmpPost1,tmpPost2,0.5);
        Posture InterPost = LinearInterpolate(fInterpDist*j, M1,AV1);
        InterPost.root_pos = positionPost[positionIndex].root_pos;
        pMotionZero->SetPosture(nCurPostureIndx, InterPost);
        nCurPostureIndx++;
    }
}

//av2
```

```
else if(i==nNumFramesInMotion1+20 )
{
    float fInterpDist = 1.0/(gap + 1.0);
    for(int j=1;j<=10;j++)
    {
        float fTemp = fInterpDist*j;
        Posture tmpPost1 = FetchPost(pMotionOne, 6);
        Posture tmpPost2 = FetchPost(pMotionTwo, nNumFramesInMotion2-2);
        AV2 = AveragePost(tmpPost1,tmpPost2,0.5);
        Posture InterPost = LinearInterpolate(fInterpDist*j, AV1,AV2);
        InterPost.root_pos = positionPost[positionIndex].root_pos;
        pMotionZero->SetPosture(nCurPostureIndx, InterPost);
        nCurPostureIndx++;
    }
}
//m21
else if(i==nNumFramesInMotion1+30 )
{
    float fInterpDist = 1.0/(gap + 1.0);
    for(int j=1;j<=10;j++)
    {
        float fTemp = fInterpDist*j;
        M21 = FetchPost(pMotionTwo,nNumFramesInMotion2-4 );
        Posture InterPost = LinearInterpolate(fInterpDist*j, AV2,M21);
        InterPost.root_pos = positionPost[positionIndex].root_pos;
        pMotionZero->SetPosture(nCurPostureIndx, InterPost);
        nCurPostureIndx++;
    }
}
//M22
else if(i==nNumFramesInMotion1+40 )
{
    float fInterpDist = 1.0/(gap + 1.0);
    for(int j=1;j<=10;j++)
    {
```

```
float fTemp = fInterpDist*j;
M22 = FetchPost(pMotionTwo, nNumFramesInMotion2-2 );
Posture InterPost = LinearInterpolate(fInterpDist*j, M21,M22);
InterPost.root_pos = positionPost[positionIndex].root_pos;
pMotionZero->SetPosture(nCurPostureIndx, InterPost);
nCurPostureIndx++;
}
}
//M22-MotionTwo
else if(i==nNumFramesInMotion1+50 )
{
float fInterpDist = 1.0/(gap + 1.0);
for(int j=1;j<=10;j++)
{
float fTemp = fInterpDist*j;
Posture InterPost = LinearInterpolate(fInterpDist*j, M22,pMotionTwo->m_pPostures[0]);
InterPost.root_pos = positionPost[positionIndex].root_pos;
pMotionZero->SetPosture(nCurPostureIndx, InterPost);
nCurPostureIndx++;
}
}

else if(i>=nNumFramesInMotion1+gap)
{
pMotionZero->SetPosture(nCurPostureIndx, pMotionTwo->m_pPostures[i-nNumFramesInMotion1 -gap]);
nCurPostureIndx++;
}
}

return pMotionZero;
}

//get posture from certain motion sequence
Posture TwoMotions::FetchPost(Motion* pMotion, int fNo)
{
```

```
    return pMotion->m_pPostures[fNo];
}

// using linear interpolation to create new posture
Posture TwoMotions::AveragePost(Posture & a, Posture &b, double s)
{
    return AveragePostures(a,b,s);
}

//LinearInterplation in Euler Angles
void TwoMotions::LinearInterpEulerAngles_TwoMotions(Motion* pMotionOne, Motion* pMotionTwo)
{
    //Assume that the first frame of the sampled motion is equal to the
    //first frame of the original motion
    //and thus equal to the first frame of interpolated motion
    pMotionZero->SetPosture(0, pMotionOne->m_pPostures[0]);

    int nCurPostureIndx = 1;

    int nNoFrame1 = pMotionOne->getNumOfFrames();
    int nNoFrame2 = pMotionTwo->getNumOfFrames();
    int gap = pMotionZero->getNumOfFrames() - nNoFrame1 - nNoFrame2;
    int nNoTotalFrame = nNoFrame1 + nNoFrame2 + gap;

    //copy two motions to new motion
    for(int i=1; i<nNoTotalFrame; i++)
    {
        if(i<nNoFrame1)
        {
            pMotionZero->SetPosture(nCurPostureIndx, pMotionOne->m_pPostures[i]);
            nCurPostureIndx++;
        }
        else if(i==nNoFrame1)
        {
            float fInterpDist = 1.0/(gap + 1.0);
            for(int j=1; j<=gap; j++)
            {
```



```
        float fTemp = fInterpDist*j;
        Posture InterPost = LinearInterpolate(fInterpDist*j,
pMotionOne->m_pPostures[nNoFrame1-1],pMotionTwo->m_pPostures[0]);

        pMotionZero->SetPosture(nCurPostureIndx, InterPost);
        nCurPostureIndx++;
    }
}

else if(i>=nNoFrame1+gap)
{
    pMotionZero->SetPosture(nCurPostureIndx, pMotionTwo->m_pPostures[i-nNoFrame1-gap]);
    nCurPostureIndx++;
}
}
}

Motion* TwoMotions::GoStraightLine()
{
    Motion* pInterpMotion=NULL;
    //Do nothing if error is set
    if (m_ErrorType != NO_ERROR_SET)
    {
        pInterpMotion = NULL;

        return NULL;
    }

    int mNumFrames = pMotionZero->getNumOfFrames();

    pInterpMotion = new Motion(mNumFrames,pMotionZero->getActor());
    pInterpMotion->SetPosture(0, pMotionZero->m_pPostures[0]);

    for(int i=1;i<mNumFrames;i++)
    {
        Posture sPosture =
```

```
StraightLine(pMotionZero->m_pPostures[0], pMotionZero->m_pPostures[6], pMotionZero->m_pPostures[i], i);
    pInterpMotion->SetPosture(i, sPosture);
}
return pInterpMotion;
}
```

```
Motion * TwoMotions::LinearInterp(int length)
{
    if (m_ErrorType != NO_ERROR_SET)
    {
        pMotionZero = NULL;
        // pMotionTwo = NULL;
        return NULL;
    }
    //Compute number of frames int the new (interpolated) motion
    int nNumFrames = 0;
    int nNoFrame1 = pMotionOne->getNumOfFrames();
    int nNoFrame2 = pMotionTwo->getNumOfFrames();
    int nNoTotalFrame = nNoFrame1+nNoFrame2-length;

    //Allocate new motion - initially set to default motion
    pMotionZero= new Motion(nNoTotalFrame, pMotionOne->getActor());

    //Assume that the first frame of the sampled motion is equal to the
    //first frame of the original motion
    //and thus equal to the first frame of interpolated motion

    pMotionZero->SetPosture(0, pMotionOne->m_pPostures[0]);

    int nCurPostureIndx = 1;

    //copy two motions to new motion
    for(int i=1; i<nNoTotalFrame; i++)
    {
        if(i<nNoFrame1-length)
        {
```

```
        pMotionZero->SetPosture (nCurPostureIndx, pMotionOne->m_pPostures [i]);
        nCurPostureIndx++;
    }
    else if (i==nNoFrame1-length )
    {
        float fInterpDist = 1.0/(length + 1.0);
        for(int j=1;j<=length;j++)
        {
            //float fTemp = fInterpDist*j;
            Posture InterPost = LinearInterpolate(fInterpDist*j,
pMotionOne->m_pPostures [i+j-1],pMotionTwo->m_pPostures [j]);

            pMotionZero->SetPosture (nCurPostureIndx, InterPost);
            nCurPostureIndx++;
        }
    }
    else if (i>=nNoFrame1)
    {
        pMotionZero->SetPosture (nCurPostureIndx, pMotionTwo->m_pPostures [i-nNoFrame1]);
        nCurPostureIndx++;
    }
}

return pMotionZero;

}

// let two motions walk in same line
void TwoMotions::MoveToSameLine()
{
    //move to zero position in AXE Z

    double iniZM1 = getRootZ(pMotionOne->m_pPostures [0]);
    double disZM1 = DistanceZ(pMotionOne->m_pPostures [0],pMotionOne->m_pPostures [pMotionOne->m_NumFrames-1]);
```

```
double averDis = disZM1/pMotionOne->m_NumFrames;

//move MotionOne to base in Z
for(int i = 0;i<pMotionOne->m_NumFrames;i++)
{
    moveToBaseZ(iniZM1,pMotionOne->m_pPostures[i]);
}

//Move MotionTwo to next to Motion One in Z
double iniZM2 = getRootZ(pMotionTwo->m_pPostures[0]);

for(i=0;i<pMotionTwo->m_NumFrames;i++)
{
    moveToBaseZ(iniZM2-disZM1-averDis*blendLength,pMotionTwo->m_pPostures[i]);
    //moveToBaseZ(iniZM2,pMotionTwo->m_pPostures[i]);
}

//In direction X move to x=0
for(i = 0;i<pMotionOne->m_NumFrames;i++)
{
    pMotionOne->m_pPostures[i].root_pos.p[0]=0;
    //moveToBaseX(iniXM1,pMotionOne->m_pPostures[i]);
}
for(i=0;i<pMotionTwo->m_NumFrames;i++)
{
    pMotionTwo->m_pPostures[i].root_pos.p[0]=0;
}
/*
double iniXM1 = getRootX(pMotionOne->m_pPostures[0]);
for(i = 0;i<pMotionOne->m_NumFrames;i++)
{
    moveToBaseX(iniXM1,pMotionOne->m_pPostures[i]);
}
//Move MotionTwo to next to Motion One
double iniXM2 = getRootX(pMotionTwo->m_pPostures[0]);
```

```
for (i=0;i<pMotionTwo->m_NumFrames;i++)
{
    moveToBaseX(iniXM2,pMotionTwo->m_pPostures[i]);
}
*/
}

Motion* TwoMotions::SetGap(int gapLength)
{
    Motion* pNewMotionTwo = new Motion( gapLength+pMotionTwo->m_NumFrames,pMotionTwo->getActor());

    for(int i=pMotionTwo->m_NumFrames-1;i>=0;i--)
        pNewMotionTwo->m_pPostures[i+gapLength] = pMotionTwo->m_pPostures[i];
    //for(i=0;i<gapLength;i++)
    //pNewMotionTwo->SetPosturesToDefault(i,pNewMotionTwo->getActor());

    pMotionTwo = pNewMotionTwo;

    int nNumFrames = pMotionOne->m_NumFrames + pMotionTwo->m_NumFrames;
    pMotionZero= new Motion(nNumFrames,pMotionOne->getActor());

    int nCurPostureIndx=1;
    pMotionZero->SetPosture(0, pMotionOne->m_pPostures[0]);
    for(i=1;i<nNumFrames;i++)
    {
        //from 0 to end of motion one
        if(i<pMotionOne->m_NumFrames )
        {
            pMotionZero->SetPosture(nCurPostureIndx, pMotionOne->m_pPostures[i]);
            nCurPostureIndx++;
        }
        else
        {
            pMotionZero->SetPosture(nCurPostureIndx, pMotionTwo->m_pPostures[i-pMotionOne->m_NumFrames]);
        }
    }
}
```

```
        nCurPostureIndx++;
    }

}

blendLength = gapLength;

return pMotionZero;

}

//get anglar velocity of the motion at certain frame
double TwoMotions::getVelocity(Motion *pMotion, double mVelocity)
{
    int mFrameNo = pMotion->m_NumFrames;

    double diff[PM_MAX_FRAMES];
    double totalDiff = 0;

    for(int i=0;i<mFrameNo-1;i++)
    {
        diff[i]=CalVelocity(pMotion->m_pPostures[i],pMotion->m_pPostures[i+1]);
        totalDiff += diff[i];
    }

    //cout<<endl<<"totoal Diff= "<<totalDiff<<endl<<endl<<endl;

    mVelocity = totalDiff/mFrameNo;
    //cout<<endl<<"v1= "<<mVelocity<<endl<<endl<<endl;

    return mVelocity;

}

//calculate the velocity of the posture
double TwoMotions::CalVelocity(Posture p1,Posture p2)
```

```
{
    double v;
    Posture tmp;

    tmp = PostureDiff(p1,p2);
    v = CalTotalDiff(tmp);
    return v;
}

void TwoMotions::generateTimeOffSetFile()
{
    //this is the method to generate time offset file
    //according to requirement
}

double TwoMotions::getV1()
{
    return mVelocity1;
}

double TwoMotions::getV2()
{
    return mVelocity2;
}

Motion* TwoMotions::getMotion1()
{
    return pMotionOne;
}

Motion* TwoMotions::getMotion2()
{
    return pMotionTwo;
}

void TwoMotions::setV1(double v1)
```

```
{
    mVelocity1 = v1;
}

void TwoMotions::setV2(double v2)
{
    mVelocity2 = v2;
}

Motion * TwoMotions::InterpSetMiddleFrames()
{//Do nothing if error is set

    if (m_ErrorType != NO_ERROR_SET)
    {
        pMotionZero = NULL;
        // pMotionTwo = NULL;
        return NULL;
    }

    //Compute number of frames int the new (interpolated) motion
    int nNumFramesInMotion1 = pMotionOne->getNumOfFrames();
    int nNumFramesInMotion2 = pMotionTwo->getNumOfFrames();
    int nNumFrames = nNumFramesInMotion1+nNumFramesInMotion2;

    //Allocate new motion - initially set to default motion
    pMotionZero= new Motion(nNumFrames,pMotionOne->getActor());

    Posture M1,AV1,AV2,M21,M22;

    int onesixthLength = blendLength/6;

    //calculate Postions inbetween ;

    vector rootPostion[5];
```



```
Posture s1 = pMotionOne->m_pPostures [nNumFramesInMotion1-2];
Posture s2 = pMotionOne->m_pPostures [nNumFramesInMotion1-1];

Posture se1 = pMotionTwo->m_pPostures [0+blendLength];
Posture se2 = pMotionTwo->m_pPostures [1+blendLength];

vector v1 = CalRootVelolity(se1,se2,60);
vector v0 = CalRootVelolity(s1,s2,60);

vector p1 = s2.root_pos;
float time;
for(int i=0;i<5;i++)
{
    time = (1.0/6)*i;

    //time can be specified manually
    /*
    if (i==0)
        time = 0;
    if (i==1)
        time = 0.2;
    if (i==2)
        time = 0.25;
    if (i==3)
        time = 0.42;
    if (i==4)
        time = 0.5;
    */

    rootPostion[i] = CalRootPostion(p1, v0,v1,time);
    cout<<rootPostion[i].p[0]<<" "<<rootPostion[i].p[1]<<" "<<rootPostion[i].p[2]<<endl;
}

int nCurPostureIndx = 1;

float fInterpDist = 1.0/(onesixthLength + 1.0);
pMotionZero->SetPosture(0, pMotionOne->m_pPostures [0]);
```

```
for(i=1;i<nNumFrames;i++ )
{
    int positionIndex = i-nNumFramesInMotion1+1;

    //from 0 to end of motion one
    if(i<nNumFramesInMotion1)
    {
        pMotionZero->SetPosture(nCurPostureIndx, pMotionOne->m_pPostures[i]);
        nCurPostureIndx++;
    }

    //MotionOne -m1
    else if(i==nNumFramesInMotion1 )
    {
        for(int j=1;j<=blendLength/6;j++)
        {
            M1 = FetchPost(pMotionOne, 5);

            //Posture InterPost;
            //InterPost.root_pos = rootPostion[0];
            //InterPost = LinearInterpolate(fInterpDist*j, pMotionOne->m_pPostures [nNumFramesInMotion1
-1],M1);
            M1.root_pos = rootPostion[0];
            Posture InterPost = LinearInterpolate(fInterpDist*j, pMotionOne->m_pPostures [nNumFramesInMotion1
-1],M1);

            //InterPost.root_pos = rootPostion[0];
            pMotionZero->SetPosture(nCurPostureIndx, InterPost);
            nCurPostureIndx++;
        }
    }

    // av1
    else if(i==nNumFramesInMotion1+1*onesixthLength )
```

```
{
    for(int j=1;j<=blendLength/6;j++)
    {
        Posture tmpPost1 = FetchPost(pMotionOne, 5);
        Posture tmpPost2 = FetchPost(pMotionTwo, nNumFramesInMotion2-15);
        AV1 = AveragePost(tmpPost1,tmpPost2,0.5);
        AV1.root_pos = rootPostion[1];
        Posture InterPost = LinearInterpolate(fInterpDist*j, M1,AV1);

        pMotionZero->SetPosture(nCurPostureIndx, InterPost);
        nCurPostureIndx++;
    }
}

//av2
else if(i==nNumFramesInMotion1+2*onesixthLength )
{
    for(int j=1;j<=blendLength/6;j++)
    {
        Posture tmpPost1 = FetchPost(pMotionOne, 10);
        Posture tmpPost2 = FetchPost(pMotionTwo, nNumFramesInMotion2-10);
        AV2 = AveragePost(tmpPost1,tmpPost2,0.5);
        AV2.root_pos = rootPostion[2];
        Posture InterPost = LinearInterpolate(fInterpDist*j, AV1,AV2);

        pMotionZero->SetPosture(nCurPostureIndx, InterPost);
        nCurPostureIndx++;
    }
}

//m21
else if(i==nNumFramesInMotion1+3*onesixthLength )
{
```

```
for(int j=1;j<=blendLength/6;j++)
{
    M21 = FetchPost(pMotionTwo, nNumFramesInMotion2-10 );
    M21.root_pos = rootPosition[3];
    Posture InterPost = LinearInterpolate(fInterpDist*j, AV2,M21);

    pMotionZero->SetPosture(nCurPostureIndx, InterPost);
    nCurPostureIndx++;
}

//M22
else if(i==nNumFramesInMotion1+4*onesixthLength )
{
    for(int j=1;j<=blendLength/6;j++)
    {
        M22 = FetchPost(pMotionTwo, nNumFramesInMotion2-5 );
        M22.root_pos = rootPosition[4];
        Posture InterPost = LinearInterpolate(fInterpDist*j, M21,M22);

        pMotionZero->SetPosture(nCurPostureIndx, InterPost);
        nCurPostureIndx++;
    }
}

//M22-MotionTwo
else if(i==nNumFramesInMotion1+5*onesixthLength )
{
    for(int j=1;j<=blendLength/6;j++)
    {
        double dist = -M22.root_pos.p[2]+pMotionTwo->m_pPostures[blendLength].root_pos.p[2];
        if (dist<0){
            dist = -1* dist;
        }
    }
}
```

```
        for(int k=blendLength;k<nNumFramesInMotion2;k++)
        {
            pMotionTwo->m_pPostures[k].root_pos[2] += 1.2*dist;
        }
    }
    Posture InterPost = LinearInterpolate(fInterpDist*j,
M22,pMotionTwo->m_pPostures[0+blendLength]);

    pMotionZero->SetPosture(nCurPostureIndx, InterPost);
    nCurPostureIndx++;
}

else if(i>=nNumFramesInMotion1+blendLength)
{
    pMotionZero->SetPosture(nCurPostureIndx, pMotionTwo->m_pPostures[i-nNumFramesInMotion1]);
    nCurPostureIndx++;
}

//move the Motion Two root position next to last frame
double x = getRootZ(pMotionZero->m_pPostures[nNumFramesInMotion1+blendLength-1]);
double y = getRootZ(pMotionZero->m_pPostures[nNumFramesInMotion1+blendLength]);

//Move MotionTwo to next to Motion One in Z
double iniZM2 = getRootZ(pMotionTwo->m_pPostures[0]);

for(i=nNumFramesInMotion1+blendLength;i<nNumFrames;i++)
{
    moveToBaseZ(y-x,pMotionZero->m_pPostures[i]);
}

//CatmullRomPosition(pMotionZero);
```

```
        CatmullRomSpline(pMotionZero);
        return pMotionZero;
    }

vector TwoMotions::CalRootVelocity(Posture p1, Posture p2, double frameRate)
{
    vector r1 = p1.root_pos;
    vector r2 = p2.root_pos;

    vector v = (r2 - r1)*frameRate;

    return v;
}

vector TwoMotions::CalUnitVector(vector v)
{
    return v/v.length();
}

vector TwoMotions::AlphaT(vector v1, vector v2, double t)
{
    vector result;
    result = (v2-v1)* t * t / 2 + v1 * t;
    return result;
}

vector TwoMotions::CalRootPosition(vector p1, vector vel1, vector vel2, double time)
{
    vector result;
    result = p1 + AlphaT(vel1, vel2, time);

    return result;
}

void TwoMotions::CatmullRomSpline(Motion *& motion)
```

```
{
    Posture v1,v2,v3,v4;
    Posture posture;

    //set parameter t by setting every 4 frames manipulated
    float t= 1.0/(4+1.0);

    for(int i=1;i<motion->m_NumFrames-6;i++){

        v1 = motion->m_pPostures[i-1];
        v2 = motion->m_pPostures[i];
        v3 = motion->m_pPostures[i+5];
        v4 = motion->m_pPostures[i+6];

        for(int j=1;j<=4;j++){

            //Posture CatmullRomInterpolate( Posture const& p1, Posture const& p2, Posture const& p3, Posture const&
            p4,float t )
            posture = CatmullRomInterpolate( v1, v2,v3,v4,t*j );

            //round float to interger frame number
            int cframe =(int) (i+(4+1)*t*j+0.001);

            motion->SetPosture(cframe,posture);
        }
    }
}
```

```
//-----  
//  Filename: motion.h  
//  Description:  
//  class that defines motion  
//  1. read an AMC file and store it in a sequence of state vector  
//  2. write an AMC file  
//  3. export to a mrdplot format for plotting the trajectories  
//  4. methods for single motion manipulation can be added  
//-----  
  
#ifndef _MOTION_H  
#define _MOTION_H  
  
#include "vector.h"  
#include "types.h"  
#include "posture.h"  
#include "skeleton.h"  
#include "posture_q.h"  
  
class Motion  
{  
    //member functions  
public:  
    //Include Actor (skeleton) ptr  
    Motion(char *amc_filename, float scale, Skeleton * pActor);  
    //Use to creating motion from AMC file  
    Motion(char *amc_filename, float scale);  
    //Use to create default motion with specified number of frames  
    Motion(int nFrameNum, Skeleton * pActor);  
    //delete motion  
    ~Motion();  
  
    // scale is a parameter to adjust the translational parameter  
    // This value should be consistent with the scale parameter used in Skeleton()  
    // The default value is 0.06  
    int readAMCfile(char* name, float scale);  
    int writeAMCfile(char* name, float scale);  
    int editAMCfile(char* name, float scale, int start, int end);  
};
```



```
//Root position at (0,0,0), orientation of each bone to (0,0,0)
void SetPosturesToDefault();//original settodefault, useless
void SetPosturesToDefault(int nFrame,Skeleton* pActor2);

//set skeleton to motion
Skeleton* getActor();
void setActor(Skeleton*);
int getNumOfFrames();

//Set posture at spesified frame
void SetPosture(int nFrameNum, Posture InPosture);
int GetPostureNum(int nFrameNum);
void SetTimeOffset(int n_offset);
Posture* GetPosture(int nFrameNum);
void SetBoneRotation(int nFrameNum, vector vRot, int nBone);
void SetRootPos(int nFrameNum, vector vPos);

//data members
public:
int m_NumFrames; //Number of frames in the motion
int offset;

//Overall number of degrees of freedom (summation of degrees of freedom for all bones)
//int m_NumDOFs;
Skeleton * pActor;
//Root position and all bone rotation angles for each frame (as read from AMC file)
Posture* m_pPostures;
posture_q* m_pPostures_q;
};

#endif
```

```
//-----  
//  Filename: motion.cpp  
//  Description:  
//  class that defines motion  
//  1. read an AMC file and store it in a sequence of state vector  
//  2. write an AMC file  
//  3. export to a mrdplot format for plotting the trajectories  
//  4. methods for single motion manipulation can be added  
//-----  
  
#include <stdio.h>  
#include <string.h>  
#include <fstream.h>  
#include <math.h>  
  
#include "skeleton.h"  
#include "motion.h"  
#include "vector.h"  
#include "quaternion.h"  
#include "posture_q.h"  
  
// a default skeleton that defines each bone's degree of freedom and the order of the data stored in the AMC  
file  
//static Skeleton actor("Skeleton.ASF", MOCAP_SCALE);  
typedef float * floatptr;  
  
Motion::Motion(int nNumFrames, Skeleton * pActor2)  
{  
    // m_NumDOFs = pActor.m_NumDOFs;  
  
    pActor = pActor2;  
    m_NumFrames = nNumFrames;  
    offset = 0;  
  
    //allocate postures array  
    m_pPostures = new Posture [m_NumFrames];
```

```
        //Set all postures to default posture
        SetPosturesToDefault(m_NumFrames,pActor);
    }

Motion::Motion(char *amc_filename, float scale,Skeleton * pActor2)
{
    pActor = pActor2;

    // m_NumDOFs = actor.m_NumDOFs;
    offset = 0;
    m_NumFrames = 0;
    m_pPostures = NULL;
    readAMCfile(amc_filename, scale);
}

Motion::Motion(char *amc_filename, float scale)
{
    // m_NumDOFs = actor.m_NumDOFs;
    offset = 0;
    m_NumFrames = 0;
    m_pPostures = NULL;
    readAMCfile(amc_filename, scale);
}

Motion::~Motion()
{
    if (m_pPostures != NULL)
        delete [] m_pPostures;
}

void Motion::SetPosturesToDefault(int nFrame,Skeleton* pActor2)
{
    //for each frame
    //int numbones = numBonesInSkel(bone[0]);
    for (int i = 0; i<nFrame; i++)
    {
```

```
//set root position to (0,0,0)
m_pPostures[i].root_pos.setValue(0.0, 0.0, 0.0);
//set each bone orientation to (0,0,0)
for (int j = 0; j < (*pActor2).getNumberOfBonesInASF(); j++)
{
    m_pPostures[i].bone_rotation[j].setValue(0.0, 0.0, 0.0);
    m_pPostures[i].bone_translation[j].setValue(0.0, 0.0, 0.0);
}
}

void Motion::setActor(Skeleton* npActor)
{
    pActor = npActor;
}

Skeleton* Motion::getActor()
{
    return pActor;
}

//Set posture at spesified frame
void Motion::SetPosture(int nFrameNum, Posture InPosture)
{
    m_pPostures[nFrameNum] = InPosture;
}

int Motion::GetPostureNum(int nFrameNum)
{
    nFrameNum += offset;

    if (nFrameNum < 0)
        return 0;
    else if (nFrameNum >= m_NumFrames)
        return m_NumFrames-1;
    else
```

```
        return nFrameNum;
    return 0;
}

void Motion::SetTimeOffset(int n_offset)
{
    offset = n_offset;
}

void Motion::SetBoneRotation(int nFrameNum, vector vRot, int nBone)
{
    m_pPostures[nFrameNum].bone_rotation[nBone] = vRot;
}

void Motion::SetRootPos(int nFrameNum, vector vPos)
{
    m_pPostures[nFrameNum].root_pos = vPos;
}

Posture* Motion::GetPosture(int nFrameNum)
{
    if (m_pPostures != NULL)
        return &m_pPostures[nFrameNum];
    else
        return NULL;
}

int Motion::readAMCfile(char* name, float scale)
{
    Bone *hroot, *bone;
    bone = hroot= (*pActor).getRoot();

    ifstream file( name, ios::in | ios::nocreate );
    if( file.fail() ) return -1;

    int n=0;
```

```
char str[2048];

// count the number of lines
while(!file.eof())
{
    file.getline(str, 2048);
    if(file.eof()) break;
    //We do not want to count empty lines
    if (strcmp(str, "") != 0)
        n++;
}

file.close();

//Compute number of frames.
//Subtract 3 to ignore the header
//There are (NUM_BONES_IN_ASF_FILE - 2) moving bones and 2 dummy bones (lhipjoint and rhipjoint)
int numbones = numBonesInSkel(bone[0]);
int movbones = movBonesInSkel(bone[0]);
n = (n-3)/((movbones) + 1);

m_NumFrames = n;

//Allocate memory for state vector
m_pPostures = new Posture [m_NumFrames];

m_pPostures_q = new posture_q[m_NumFrames];

file.open( name );

// skip the header
while (1)
{
    file >> str;
    if(strcmp(str, ":DEGREES") == 0) break;
}

int frame_num;
```

```
float x, y, z;
int i, bone_idx, state_idx;

for(i=0; i<m_NumFrames; i++)
{
    //read frame number
    file >> frame_num;
    x=y=z=0;

    //There are (NUM_BONES_IN_ASF_FILE - 2) moving bones and 2 dummy bones (lhipjoint and rhipjoint)
    for( int j=0; j<movbones; j++ )
    {
        //read bone name
        file >> str;

        //Convert to corresponding integer
        for( bone_idx = 0; bone_idx < numbones; bone_idx++ )
        //    if( strcmp( str, AsfPartName[bone_idx] ) == 0 )
        //        if( strcmp( str, pActor->idx2name(bone_idx) ) == 0 )

            break;

        //init rotation angles for this bone to (0, 0, 0)
        m_pPostures[i].bone_rotation[bone_idx].setValue(0.0, 0.0, 0.0);

        for(int x = 0; x < bone[bone_idx].dof; x++)
        {
            float tmp;
            file >> tmp;
            // printf("%d %f\n",bone[bone_idx].dofo[x],tmp);
            switch (bone[bone_idx].dofo[x])
            {
                case 0:
                    printf("FATAL ERROR in bone %d not found %d\n",bone_idx,x);
                    x = bone[bone_idx].dof;
                    break;
                case 1:
```

```
        m_pPostures[i].bone_rotation[bone_idx].p[0] = tmp;
        break;
    case 2:
        m_pPostures[i].bone_rotation[bone_idx].p[1] = tmp;
        break;
    case 3:
        m_pPostures[i].bone_rotation[bone_idx].p[2] = tmp;
        break;
    case 4:
        m_pPostures[i].bone_translation[bone_idx].p[0] = tmp * scale;
        break;
    case 5:
        m_pPostures[i].bone_translation[bone_idx].p[1] = tmp * scale;
        break;
    case 6:
        m_pPostures[i].bone_translation[bone_idx].p[2] = tmp * scale;
        break;
    case 7:
        m_pPostures[i].bone_length[bone_idx].p[0] = tmp; // * scale;
        break;
    }

}

if( strcmp( str, "root" ) == 0 )
{
    m_pPostures[i].root_pos.p[0] = m_pPostures[i].bone_translation[0].p[0]; // * scale;
    m_pPostures[i].root_pos.p[1] = m_pPostures[i].bone_translation[0].p[1]; // * scale;
    m_pPostures[i].root_pos.p[2] = m_pPostures[i].bone_translation[0].p[2]; // * scale;
    //convert to quaternion

    m_pPostures_q[i].root_pos_q =
    CalEulerAngleToQuat(m_pPostures[i].bone_translation[0].p[0], m_pPostures[i].bone_translation[0].p[1], m_pPostures[i].bone_translation[0].p[2]);
}
//convert joint angles to quaternions
m_pPostures_q[i].bone_rotation_q[bone_idx] =
CalEulerAngleToQuat(m_pPostures[i].bone_rotation[bone_idx].p[0], m_pPostures[i].bone_rotation[bone_idx].p[1]
```



```
,m_pPostures[i].bone_rotation[bone_idx].p[2]);

    // read joint angles, including root orientation
}

file.close();
printf("%d samples in '%s' are read.\n", n, name);
return n;
}

int Motion::writeAMCfile(char *filename, float scale)
{
    int f, n, j, d;
    Bone *bone;
    bone=(*pActor).getRoot();

    ofstream os(filename);
    if(os.fail()) return -1;

    // header lines
    os << "#Unknow ASF file" << endl;
    os << ":FULLY-SPECIFIED" << endl;
    os << ":DEGREES" << endl;
    int numbones = numBonesInSkel(bone[0]);

    for(f=0; f < m_NumFrames; f++)
    {
        os << f+1 <<endl;
        os << "root " << m_pPostures[f].root_pos.p[0]/scale << " "
            << m_pPostures[f].root_pos.p[1]/scale << " "
            << m_pPostures[f].root_pos.p[2]/scale << " "
```

```
                << m_pPostures[f].bone_rotation[root].p[0] << " "
                << m_pPostures[f].bone_rotation[root].p[1] << " "
                << m_pPostures[f].bone_rotation[root].p[2] ;

n=6;

for(j = 2; j < numbones; j++)
{
    //output bone name
    if(bone[j].dof != 0)
//      os << endl << AsfPartName[j];
      os << endl << pActor->idx2name(j);

    //output bone rotation angles
    if(bone[j].dofx == 1)
      os << " " << m_pPostures[f].bone_rotation[j].p[0];

    if(bone[j].dofy == 1)
      os << " " << m_pPostures[f].bone_rotation[j].p[1];

    if(bone[j].dofz == 1)
      os << " " << m_pPostures[f].bone_rotation[j].p[2];
    }
  os << endl;
}

os.close();
printf("Write %d samples to '%s' \n", m_NumFrames, filename);
return 0;
}

int Motion::editAMCfile(char* filename, float scale,int start,int end)
{
  int f, n, j, d;
  Bone *bone;
  bone=(*pActor).getRoot();

  ofstream os(filename);
```

```
if(os.fail()) return -1;

// header lines
os << "#Unknow ASF file" << endl;
os << ":FULLY-SPECIFIED" << endl;
os << ":DEGREES" << endl;
int numbones = numBonesInSkel(bone[0]);

for(f=start-1; f < end-1; f++)
{
    os << f+1-start <<endl;
    os << "root " << m_pPostures[f].root_pos.p[0]/scale << " "
        << m_pPostures[f].root_pos.p[1]/scale << " "
        << m_pPostures[f].root_pos.p[2]/scale << " "
        << m_pPostures[f].bone_rotation[root].p[0] << " "
        << m_pPostures[f].bone_rotation[root].p[1] << " "
        << m_pPostures[f].bone_rotation[root].p[2] ;

    n=6;

    for(j = 2; j < numbones; j++)
    {

        //output bone name
        if(bone[j].dof != 0)
        //    os << endl << AsfPartName[j];
        os << endl << pActor->idx2name(j);

        //output bone rotation angles
        if(bone[j].dofx == 1)
            os << " " << m_pPostures[f].bone_rotation[j].p[0];

        if(bone[j].dofy == 1)
            os << " " << m_pPostures[f].bone_rotation[j].p[1];

        if(bone[j].dofz == 1)
            os << " " << m_pPostures[f].bone_rotation[j].p[2];

    }
}
```

motion.cpp

```
        os << endl;
    }

    os.close();
    printf("Write %d samples to '%s' \n", end-start,
    return 0;
}

int Motion::getNumOfFrames()
{
    return m_NumFrames;
}
```

lyit

Institiúid Teicneolaíochta Leitir Ceannainn
Letterkenny Institute of Technology

filename);

lyit

Institiúid Teicneolaíochta Leitir Ceanaínn
Letterkenny Institute of Technology

player.h

```
//-----  
//  Filename: player.h  
//  Description:  
//  All the user interface functions .  
//-----  
  
#ifndef _PLAYER_H  
#define _PLAYER_H  
  
#include <FL/Fl_Gl_Window.H>  
  
class Player_Gl_Window : public Fl_Gl_Window  
{  
private:  
    int handle_mouse(int event);  
    int handle_key(int event);  
  
public:  
    inline Player_Gl_Window(int x, int y, int w, int h, const char *l=0) :  
        Fl_Gl_Window(x, y, w, h, l) {};  
  
    /* This is an overloading of a Fl_Gl_Window call. It is called  
       whenever a window needs refreshing. */  
    void draw();  
  
    /* This is an overloading of a Window call. It is  
       called whenever an event happens inside the space  
       taken up by the Anim_Gl_Window. */  
    int handle(int event);  
  
    /* Provided Save Function */  
    void save(char *);  
};  
  
typedef struct _MouseT {  
    int button;
```

player.h

```
int x;  
int y;  
} MouseT;
```

```
typedef struct _CameraT {  
    double zoom;  
    double tw;  
    double el;  
    double az;  
    double tx;  
    double ty;  
    double tz;  
    double atx;  
    double aty;  
    double atz;  
} CameraT;
```

```
void gl_init();  
void light_init();  
void display();  
static void error_check(int loc);  
#endif
```

lyit | **Institiúid Teicneolaíochta Leitir Ceanaínn**
Letterkenny Institute of Technology


```
//-----  
//  Filename: player.cpp  
//  Description:  
//  All the user interface functions .  
//-----  
  
#ifdef WIN32  
#include <FL/gl.h>  
#endif  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <string>  
#include <cstring>           // Add string functionality  
#include <fstream.h>  
#include <assert.h>  
#include <math.h>  
#include <process.h>  
  
#include <GL/gl.h>           // Header so that you can use GL routines (MESA)  
#include <GL/glu.h>          // some OpenGL extensions  
#include <FL/glut.H>         // GLUT for use with FLTK  
#include <FL/fl_file_chooser.H> // Allow a file chooser for save.  
  
#include "player.h"  
#include "interface.h"       // UI framework built by FLTK (using fluid)  
#include "pic.h"             // for saving jpeg pictures.  
#include "transform.h"       // utility functions for vector and matrix transformation  
#include "display.h"  
#include "interpolator.h"  
#include "video_texture.h"  
  
#include "InitMotionData.h"  
#include "twoMotions.h"  
#include <sstream>  
  
using namespace std;
```

```
enum {OFF, ON};

static Display displayer;
// Actor info as read from ASF file
static Skeleton *pActor = NULL;
// Set to true if actor exists
static bool bActorExist = false;
// Motion information as read from AMC file
static Motion *pSampledMotion = NULL;
static Motion *pInterpMotion = NULL;
// Interpolated Motion

static Motion * pMotion1= NULL;
static Motion * pMotion2= NULL;

static int nFrameNum, nFrameInc=1;
// Current frame and frame increment

static Fl_Window *form=NULL;
// Global form
static MouseT mouse;
// Keeping track of mouse input
static CameraT camera;
// Structure about camera setting

static int Play=OFF, Rewind=OFF;
// Some Flags for player
static int Repeat=OFF, Record=OFF;

static int PlayInterpMotion=ON;
// Flag which decides which motion to play (pSampledMotion or pInterpMotion)
static int Background=ON, Light=OFF;
// Flags indicating if the object exists

static char *Record_filename;
// Recording file name
```

```
static int recmode = 0;
static int piccount=0;
static char * argv2;
static int maxFrames=0;

static bool changeMotion = false;

static void draw_triad()
{
    glBegin(GL_LINES);

    /* draw x axis in red, y axis in green, z axis in blue */
    glColor3f(1., .2, .2);
    glVertex3f(0., 0., 0.);
    glVertex3f(1., 0., 0.);

    glColor3f(.2, 1., .2);
    glVertex3f(0., 0., 0.);
    glVertex3f(0., 1., 0.);

    glColor3f(.2, .2, 1.);
    glVertex3f(0., 0., 0.);
    glVertex3f(0., 0., 1.);

    glEnd();
}

//Draw checker board ground plane
static void draw_ground()
{
    float i, j;
    int count = 0;

    GLfloat white4[] = {.4, .4, .4, 1.};
    GLfloat white1[] = {.1, .1, .1, 1.};
    GLfloat green5[] = {0., .5, 0., 1.};
    GLfloat green2[] = {0., .2, 0., 1.};
```

is in blue */

lyit | **Institiúid Teicneolaíochta Leitir Ceanaínn**
Letterkenny Institute of Technology

```
GLfloat black[] = {0., 0., 0., 1.};
GLfloat mat_shininess[] = {7.}; /* Phong exponent */

glBegin(GL_QUADS);

for(i=-15.;i<=15.;i+=1)
{
    for(j=-15.;j<=15.;j+=1)
    {
        if((count%2) == 0)
        {
            glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, black);
            glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, white4);
            // glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, white1);
            // glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, mat_shininess);
            glColor3f(.6, .6, .6);
        }
        else
        {
            glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, black);
            glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, green5);
            // glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, green2);
            // glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, mat_shininess);
            glColor3f(.8, .8, .8);
        }

        glNormal3f(0.,0.,1.);

        glVertex3f(j, 0, i);
        glVertex3f(j, 0, i+1);
        glVertex3f(j+1,0, i+1);
        glVertex3f(j+1,0, i);
        count++;
    }
}

glEnd();
}
```

```
void cameraView(void)
{
    glTranslated(camera.tx, camera.ty, camera.tz);
    glTranslated(camera.atx, camera.aty, camera.atz);

    glRotated(-camera.tw, 0.0, 1.0, 0.0);
    glRotated(-camera.el, 1.0, 0.0, 0.0);
    glRotated(camera.az, 0.0, 1.0, 0.0);

    glTranslated(-camera.atx, -camera.aty, -camera.atz);
    glScaled(camera.zoom, camera.zoom, camera.zoom);
}

/*
 * redisplay() is called by Player_Gl_Window::draw().
 *
 * The display is double buffered, and FLTK swap buffers when
 * Player_Gl_Window::draw() returns. The GL context associated with this
 * instance of Player_Gl_Window is set to be the current context by FLTK
 * when it calls draw().
 */
static void redisplay()
{
    if(Light) glEnable(GL_LIGHTING);
    else glDisable(GL_LIGHTING);

    /* clear image buffer to black */
    glClearColor(0, 0, 0, 0);
    glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT); /* clear image, zbuf */

    glPushMatrix();          /* save current transform matrix */

    cameraView();
}
```

```
glLineWidth(2.);
if (Background)
{
    draw_triad();
    draw_ground();
}

if (bActorExist) displayer.show();

glPopMatrix();      /* restore current transform matrix */
}

/* Callbacks from form. */
void redisplay_proc(Fl_Light_Button *obj, long val)
{
    Light = light_button->value();
    Background = background_button->value();
    glwindow->redraw();
}

//Interpolate loaded motion using linear interpolation
void initMotionData_callback(Fl_Button *button, void *)
{
    char *filename;
    char *cycleFileName;
    Motion * pMotion;

    if(button==initMotionData_button)
    {
        filename = fl_file_chooser("Select filename","*.ASF","");

        if(filename != NULL)
        {
            //Remove old actor
            //if(pActor != NULL)
```

```
        // delete pActor;
        //Read skeleton from asf file
        pActor = new Skeleton(filename, MOCAP_SCALE);

        bActorExist = true;
        glwindow->redraw();
    }
}

if(button==initMotionData_button)
{
    if (bActorExist == true)
    {
        filename = fl_file_chooser("Select filename", "*.AMC", "");
        if(filename != NULL)
        {

            pMotion = new Motion(filename, MOCAP_SCALE,pActor);

            InitMotionData *mInitM = new InitMotionData(pMotion);
            int endFrame = mInitM->GetFullCycle();

            //a file name for cycle motion file
            string s1;
            s1 = "cycle";
            string s2(filename);
            cycleFileName= (char*)s2.insert(0,s1).c_str();
            pMotion->editAMCfile(cycleFileName,MOCAP_SCALE,START,endFrame);
            //debug make a fine motion
            //pMotion->editAMCfile("Run02.amc",MOCAP_SCALE,14,140);
            mInitM->regenerateMotion(SCALE,cycleFileName);

        }
    }
}
bActorExist = false;
```



```
}
void load_callback(Fl_Button *button, void *)
{
    char *filename;

    if(button==loadActor_button)
    {
        filename = fl_file_chooser("Select filename","*.ASF","");
        if(filename != NULL)
        {
            //Remove old actor
            if(pActor != NULL)
                delete pActor;
            //Read skeleton from asf file
            pActor = new Skeleton(filename, MOCAP_SCALE);

            //Set the rotations for all bones in their local coordinate system to 0
            //Set root position to (0, 0, 0)
            pActor->setBasePosture();
            displayer.loadActor(pActor);
            bActorExist = true;
            glwindow->redraw();
        }
    }

    if(button==loadMotion_button)
    {
        if (bActorExist == true)
        {
            filename = fl_file_chooser("Select filename","*.AMC","");
            if(filename != NULL)
            {
                //delete old motion if any
                if (pSampledMotion != NULL)
                {
                    delete pSampledMotion;
                    pSampledMotion = NULL;
                }
            }
        }
    }
}
```

```
    if (pInterpMotion != NULL)
    {
        delete pInterpMotion;
        pInterpMotion = NULL;
    }

    //Read motion (.amc) file and create a motion
    pSampledMotion = new Motion(filename, MOCAP_SCALE,pActor);

    //set sampled motion for display
    displayer.loadMotion(pSampledMotion);

    //Tell actor to perform the first pose ( first posture )
    pActor->setPosture(displayer.m_pMotion->m_pPostures[0]);
    maxFrames = 0;
    if ( (displayer.m_pMotion[displayer.numActors-1]->m_NumFrames - 1) > maxFrames)
    {
        maxFrames = (displayer.m_pMotion[displayer.numActors-1]->m_NumFrames - 1);
        frame_slider->maximum((double)maxFrames+1);
    }
    nFrameNum=(int) frame_slider->value() -1;

    // display
    for (int i = 0; i < displayer.numActors; i++)

        displayer.m_pActor[i]->setPosture(displayer.m_pMotion[i]->m_pPostures[displayer.m_pMotion[i]->GetPosture
Num(nFrameNum)]);
        Fl::flush();
        glwindow->redraw();
    }
}
glwindow->redraw();
}
```

```
void twoMotions_callback(Fl_Button *button, void *)
{
    char *filename;
    char *resultFileName;

    if(button==twoMotions_button)
    {
        filename = fl_file_chooser("Select filename","*.ASF","");

        if(filename != NULL)
        {
            //Remove old actor
            //if(pActor != NULL)
            // delete pActor;
            //Read skeleton from asf file
            pActor = new Skeleton(filename, MOCAP_SCALE);

            bActorExist = true;
            glwindow->redraw();
        }
    }

    if(button==twoMotions_button)
    {
        if (bActorExist == true)
        {
            filename = fl_file_chooser("Select filename","*.AMC","");
            if(filename != NULL)
            {
                pMotion1 = new Motion(filename, MOCAP_SCALE,pActor);
            }
        }
    }
}

string str1(filename);

if(button==twoMotions_button)
```

```
{
    if (bActorExist == true)
    {
        filename = fl_file_chooser("Select filename", "*.AMC", "");
        if(filename != NULL)
        {
            pMotion2 = new Motion(filename, MOCAP_SCALE, pActor);
        }
    }
}
string tmpstr1 = str1;

for(int i=1;i<=6;i++)
{
    str1="";
    str1 = tmpstr1;

    Motion * dispMotion = NULL;
    TwoMotions* obj = new TwoMotions(pMotion1, pMotion2);
    //get two motions Velocity
    //obj->setV1(obj->getVelocity(obj->getMotion1(), obj->getV1()));
    //obj->setV2(obj->getVelocity(obj->getMotion2(), obj->getV2()));

    string str2(filename);

    //-----
    string frameNumber="";
    stringstream ss;
    ss<<6*i;
    ss>>frameNumber;

    str1+="____";
    str1+=frameNumber;
    str1+="____";

    resultFileName = (char*)str2.insert(0, str1).c_str();
}
```

```
//set the blending length between two motion
dispMotion = obj->SetGap(6*i);

//move two motions to a straight line.
obj->MoveToSameLine();

//linear interpolation by two postures
//dispMotion = (Motion*)obj->InterpolateTwoMotions(15);

//LinearInterpolation by 2 method
//dispMotion = obj->LinearInterp(5);

//
dispMotion = (Motion*)obj->InterpSetMiddleFrames();

dispMotion->writeAMCfile(resultFileName,MOCAP_SCALE);

bActorExist = false;
}

void change_callback(Fl_Button *button, void *)
{
    if (button == change_button)
    {
        changeMotion = true;
    }
}

void save_callback(Fl_Button *button, void *)
{
    //char *filename;
```

```
    if(button==save_button)
        glwindow->save(fl_file_chooser("Save to Jpeg File", "*.jpg", ""));
}

void play_callback(Fl_Button *button, void *)
{
    if(displayer.m_pMotion[0] != NULL)
    {
        if(button==play_button) { Play=ON; Rewind=OFF; }
        if(button==pause_button){ Play=OFF; Repeat=OFF; }
        if(button==repeat_button) { Rewind=OFF; Play=ON; Repeat=ON; }
        if(button==rewind_button) { Rewind=ON; Play=OFF; Repeat=OFF; }
    }
}

void record_callback(Fl_Light_Button *button, void *)
{
    int current_state = (int) button->value();

    if(Play == OFF)
    {
        if(Record == OFF && current_state == ON)
        {
            Record_filename = fl_file_chooser("Save Animation to Jpeg Files", "", "");
            if(Record_filename != NULL)
                Record = ON;
        }
        if(Record == ON && current_state == OFF)
            Record = OFF;
    }
    button->value(Record);
}

void idle(void*)
{
```

```
if (displayer.m_pMotion[0] != NULL)
{
    if (Rewind==ON)
    {
        nFrameNum=0;
        for (int i = 0; i < displayer.numActors; i++){

            displayer.m_pActor[i]->setPosture(displayer.m_pMotion[i]->m_pPostures[displayer.m_pMotion[i]->GetPosture
Num(nFrameNum)]);

        }
        Rewind=OFF;
    }

    if (Play==ON)
    {
        if (nFrameNum >= maxFrames)
        {
            if (Repeat == ON)
            {
                nFrameNum=0;
                for (int i = 0; i < displayer.numActors; i++){

                    displayer.m_pActor[i]->setPosture(displayer.m_pMotion[i]->m_pPostures[displayer.m_pMotion[i]->GetPosture
Num(nFrameNum)]);

                }
            }
            else
            {
                for (int i = 0; i < displayer.numActors; i++){

                    displayer.m_pActor[i]->setPosture(displayer.m_pMotion[i]->m_pPostures[displayer.m_pMotion[i]->GetPosture
Num(nFrameNum)]);

                }
            }
        }
    }
}
```

```
        else
            for (int i = 0; i < displayer.numActors; i++){

displayer.m_pActor[i]->setPosture(displayer.m_pMotion[i]->m_pPostures[displayer.m_pMotion[i]->GetPostureNum
(nFrameNum)]);

        }

        if (Record==ON)
            glwindow->save(Record_filename);

        if (nFrameNum < maxFrames)
            nFrameNum += nFrameInc;
    }

    if (changeMotion==false)
    {
        frame_slider->value((double)nFrameNum+1);

        glwindow->redraw();
    }

}

void fslider_callback(Fl_Value_Slider *slider, long val)
{
    if (displayer.m_pMotion[0] != NULL)
    {
        if (displayer.m_pMotion[0]->m_NumFrames > 0)
        {
            nFrameNum=(int) frame_slider->value()-1;
            for (int i = 0; i < displayer.numActors; i++)
//                if (displayer.m_pMotion[i]->m_NumFrames > 0)
```



```
        displayer.m_pActor[i]->setPosture(displayer.m_pMotion[i]->m_pPostures[displayer.m_pMotion[i]->GetPosture
Num(nFrameNum)]);
        Fl::flush();
        glwindow->redraw();
    }
}

// locate rotation center at the (root.x, 0, root.z)
void locate_callback(Fl_Button *obj, void *)
{
    if(bActorExist && displayer.m_pMotion[0] != NULL)
    {
        camera.zoom = 1;
        camera.atx = pActor->m_RootPos[0];
        camera.aty = 0;
        camera.atz = pActor->m_RootPos[2];
    }
    glwindow->redraw();
}

void valueIn_callback(Fl_Value_Input *obj, void *)
{
    displayer.m_SpotJoint = (int) joint_idx->value();
    nFrameInc = (int) fsteps->value();
    glwindow->redraw();
}

void sub_callback(Fl_Value_Input *obj, void*)
{
    int subnum;
    subnum = (int)sub_input->value();
    if (subnum < 0) sub_input->value(0);
    else if (subnum > displayer.numActors-1) sub_input->value(displayer.numActors-1);
    else
    {
        // Change values of other inputs to match subj num
    }
}
```

```
    dt_input->value(displayer.m_pMotion[subnum]->offset);
    tx_input->value(displayer.m_pActor[subnum]->tx);
    ty_input->value(displayer.m_pActor[subnum]->ty);
    tz_input->value(displayer.m_pActor[subnum]->tz);
    rx_input->value(displayer.m_pActor[subnum]->rx);
    ry_input->value(displayer.m_pActor[subnum]->ry);
    rz_input->value(displayer.m_pActor[subnum]->rz);
}
glwindow->redraw();
}

void dt_callback(Fl_Value_Input *obj, void*)
{
    int subnum,max = 0;
    subnum = (int)sub_input->value();
    if (subnum < displayer.numActors && subnum >= 0)
    {
        displayer.m_pMotion[subnum]->SetTimeOffset((int)dt_input->value());
        printf("Shifting subject %d by %d\n",subnum,(int)dt_input->value());
        for (int i = 0; i < displayer.numActors; i++)
        {
            if ((displayer.m_pMotion[i]->m_NumFrames - 1 - displayer.m_pMotion[i]->offset) > max)
                max = (displayer.m_pMotion[i]->m_NumFrames - 1 - displayer.m_pMotion[i]->offset);
        }
        maxFrames = max;
        frame_slider->maximum((double)maxFrames+1);

        displayer.m_pActor[subnum]->setPosture(displayer.m_pMotion[subnum]->m_Postures[displayer.m_pMotion[subnum]->GetPostureNum(nFrameNum)]);
    }
    glwindow->redraw();
}

void tx_callback(Fl_Value_Input *obj, void*)
{
    int subnum = 0;
    subnum = (int)sub_input->value();
```

```
    if (subnum < displayer.numActors && subnum >= 0)
    {
        displayer.m_pActor[subnum]->tx = (int)tx_input->value();
    }
    glwindow->redraw();
}

void ty_callback(Fl_Value_Input *obj, void*)
{
    int subnum = 0;
    subnum = (int)sub_input->value();
    if (subnum < displayer.numActors && subnum >= 0)
    {
        displayer.m_pActor[subnum]->ty = (int)ty_input->value();
    }
    glwindow->redraw();
}

void tz_callback(Fl_Value_Input *obj, void*)
{
    int subnum = 0;
    subnum = (int)sub_input->value();
    if (subnum < displayer.numActors && subnum >= 0)
    {
        displayer.m_pActor[subnum]->tz = (int)tz_input->value();
    }
    glwindow->redraw();
}

void rx_callback(Fl_Value_Input *obj, void*)
{
    int subnum = 0;
    subnum = (int)sub_input->value();
    if (subnum < displayer.numActors && subnum >= 0)
    {
        displayer.m_pActor[subnum]->rx = (int)rx_input->value();
    }
    glwindow->redraw();
}
```

```
}

void ry_callback(Fl_Value_Input *obj, void*)
{
    int subnum = 0;
    subnum = (int)sub_input->value();
    if (subnum < displayer.numActors && subnum >= 0)
    {
        displayer.m_pActor[subnum]->ry = (int)ry_input->value();
    }
    glwindow->redraw();
}

void rz_callback(Fl_Value_Input *obj, void*)
{
    int subnum = 0;
    subnum = (int)sub_input->value();
    if (subnum < displayer.numActors && subnum >= 0)
    {
        displayer.m_pActor[subnum]->rz = (int)rz_input->value();
    }
    glwindow->redraw();
}

void exit_callback(Fl_Button *obj, long val)
{
    //DEBUG: uncomment
    exit(1);
}

void light_init()
{
    /* set up OpenGL to do lighting
```

```
* we've set up three lights */

/* set material properties */
GLfloat white8[] = {.8, .8, .8, 1.};
GLfloat white2[] = {.2, .2, .2, 1.};
GLfloat black[] = {0., 0., 0., 1.};
GLfloat mat_shininess[] = {50.}; /* Phong exponent */

GLfloat light0_position[] = {-25., 25., 25., 0.}; /* directional light (w=0) */
GLfloat white[] = {11., 11., 11., 5.};

GLfloat light1_position[] = {-25., 25., -25., 0.};
GLfloat red[] = {1., .3, .3, 5.};

GLfloat light2_position[] = {25., 25., -5., 0.};
GLfloat blue[] = {.3, .4, 1., 25.};

glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, white2); /* no ambient */
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, white8);
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, white2);
glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, mat_shininess);

/* set up several lights */
/* one white light for the front, red and blue lights for the left & top */

glLightfv(GL_LIGHT0, GL_POSITION, light0_position);
glLightfv(GL_LIGHT0, GL_DIFFUSE, white);
glLightfv(GL_LIGHT0, GL_SPECULAR, white);
glEnable(GL_LIGHT0);

glLightfv(GL_LIGHT1, GL_POSITION, light1_position);
glLightfv(GL_LIGHT1, GL_DIFFUSE, red);
glLightfv(GL_LIGHT1, GL_SPECULAR, red);
glEnable(GL_LIGHT1);

glLightfv(GL_LIGHT2, GL_POSITION, light2_position);
glLightfv(GL_LIGHT2, GL_DIFFUSE, blue);
glLightfv(GL_LIGHT2, GL_SPECULAR, blue);
```

```
    glEnable(GL_LIGHT2);

//mstevens
    GLfloat light3_position[] = {0., -25., 0., 0.6};
    glLightfv(GL_LIGHT3, GL_POSITION, light3_position);
    glLightfv(GL_LIGHT3, GL_DIFFUSE, white);
    glLightfv(GL_LIGHT3, GL_SPECULAR, white);
    glEnable(GL_LIGHT3);

    glEnable(GL_NORMALIZE); /* normalize normal vectors */
    glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE); /* two-sided lighting*/

    /* do the following when you want to turn on lighting */
    if(Light) glEnable(GL_LIGHTING);
    else glDisable(GL_LIGHTING);
}

static void error_check(int loc)
{
    /* this routine checks to see if OpenGL errors have occurred recently */
    GLenum e;

    while ((e = glGetError()) != GL_NO_ERROR)
        fprintf(stderr, "Error: %s before location %d\n",
            gluErrorString(e), loc);
}

void gl_init()
{
    int red_bits, green_bits, blue_bits;
    struct {GLint x, y, width, height;} viewport;
    glEnable(GL_DEPTH_TEST); /* turn on z-buffer */

    glGetIntegerv(GL_RED_BITS, &red_bits);
    glGetIntegerv(GL_GREEN_BITS, &green_bits);
    glGetIntegerv(GL_BLUE_BITS, &blue_bits);
}
```

```
glGetIntegerv(GL_VIEWPORT, &viewport.x);
printf("OpenGL window has %d bits red, %d green, %d blue; viewport is %dx%d\n",
red_bits, green_bits, blue_bits, viewport.width, viewport.height);

/* setup perspective camera with OpenGL */
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(/*vertical field of view*/ 45.,
/*aspect ratio*/ (double) viewport.width/viewport.height,
/*znear*/ .1, /*zfar*/ 50.);

/* from here on we're setting modeling transformations */
glMatrixMode(GL_MODELVIEW);

//Move away from center
glTranslatef(0., 0., -5.);

camera.zoom = 1;

camera.tw = 0;
camera.el = -15;
camera.az = -25;

camera.atx = 0;
camera.aty = 0;
camera.atz = 0;
}

/*****
* Define the methods for glwindow, a subset of Fl_Gl_Window.
*****/

/*
* Handle keyboard and mouse events. Don't make any OpenGL calls here;
* the GL Context is not set! Make the calls in redisplay() and call
* the redraw() method to cause FLTK to set up the context and call draw().
* See the FLTK documentation under "Using OpenGL in FLTK" for additional
```

```
* tricks and tips.
*/
int Player_Gl_Window::handle(int event)
{
    int handled = 1;
    static int prev_x, prev_y;
    int delta_x=0, delta_y=0;
    float ev_x, ev_y;

    switch(event) {
    case FL_RELEASE:
        mouse.x = (Fl::event_x());
        mouse.y = (Fl::event_y());
        mouse.button = 0;
        break;
    case FL_PUSH:
        mouse.x = (Fl::event_x());
        mouse.y = (Fl::event_y());
        mouse.button = (Fl::event_button());
        break;
    case FL_DRAG:
        mouse.x = (Fl::event_x());
        mouse.y = (Fl::event_y());
        delta_x=mouse.x-prev_x;
        delta_y=mouse.y-prev_y;

        if(mouse.button == 1)
        {
            if(abs(delta_x) > abs(delta_y))
                camera.az += (GLdouble) (delta_x);
            else
                camera.el -= (GLdouble) (delta_y);
        }
        else if(mouse.button==2)
        {
            if(abs(delta_y) > abs(delta_x))
            {
```

lyit | Institiúid Teicneolaíochta Leitir Ceanáin
Letterkenny Institute of Technology

```
    glScalef(1+delta_y/100.,1+delta_y/100.,1+delta_y/100.);
    // camera.zoom -= (GLdouble) delta_y/100.0;
    // if(camera.zoom < 0.) camera.zoom = 0;
}
}
else if(mouse.button==3){
    //camera.tx += (GLdouble) delta_x/10.0;
    //camera.tz -= (GLdouble) delta_y/10.0; //FLTK's origin is at the left_top corner

    camera.tx += (GLdouble) cos(camera.az/180.0*3.141)*delta_x/10.0;
    camera.tz += (GLdouble) sin(camera.az/180.0*3.141)*delta_x/10.0;
    camera.ty -= (GLdouble) delta_y/10.0; //FLTK's origin is at the left_top corner

    camera.atx = -camera.tx;
    camera.aty = -camera.ty;
    camera.atz = -camera.tz;
}
break;
case FL_KEYBOARD:

    switch (Fl::event_key()) {
    case 'c':
        label("letter c");

        break;
    case 'Q':
        //cout<< "Q was pressed!!"<<endl;
    case 65307:
        exit(1);
    }
    break;
default:
    // pass other events to the base class...
    handled= Fl_Gl_Window::handle(event);
}

prev_x=mouse.x;
prev_y=mouse.y;
```

```
    glwindow->redraw();

    return (handled); // Returning one acknowledges that we handled this event
}

/*
   Prewritten Save Function
*/
void Player_Gl_Window::save (char *filename)
{
    int i;
    int j;
    static char anim_filename[512];
    static Pic *in = NULL;

    sprintf(anim_filename, "%05d.jpg", piccount++);
    if (filename == NULL) return;

    //Allocate a picture buffer.
    if(in == NULL) in = pic_alloc(640,480,3,NULL);

    printf("File to save to: %s\n", anim_filename);

    for (i=479; i>=0; i--)
    {
        glReadPixels(0,479-i,640,1,GL_RGB, GL_UNSIGNED_BYTE,
                    &in->pix[i*in->nx*in->bpp]);
    }

    if (jpeg_write(anim_filename, in))
        printf("%s saved Successfully\n", anim_filename);
    else
        printf("Error in Saving\n");
}

/*
```

```
    Prewritten Draw Function.
*/
void Player_Gl_Window::draw ()
{
    //Upon setup of the window (or when Fl_Gl_Window->invalidate is called),
    //the set of functions inside the if block are executed.
    if (!valid())
    {
        /*
        if (fopen("Skeleton.ASF", "r") == NULL)
        {
            printf("Program can't run without 'Skeleton.ASF'!\n"
                "Please make sure you place a 'Skeleton.ASF' file to working directory.\n");
            exit(1);
        }*/
        gl_init();
        light_init();
    }

    //Redisplay the screen then put the proper buffer on the screen.
    redisplay();
}

int main(int argc, char **argv)
{
    /* initialize form, sliders and buttons*/
    form = make_window();

    light_button->value(Light);
    background_button->value(Background);
    record_button->value(Record);

    frame_slider->value(1);

    /*show form, and do initial draw of model */
    form->show();
    glwindow->show(); /* glwindow is initialized when the form is built */
}
```

```
if (argc > 2)
{
    char *filename;

    if(1==1)
    {
        filename = argv[1];
        if(filename != NULL)
        {
            //Remove old actor
            if(pActor != NULL)
                delete pActor;
            //Read skeleton from asf file
            pActor = new Skeleton(filename, MOCAP_SCALE);

            //Set the rotations for all bones in their local coordinate system to 0
            //Set root position to (0, 0, 0)
            pActor->setBasePosture();
            displayer.loadActor(pActor);
            bActorExist = true;
        }
    }

    if(1==1)
    {
        if (bActorExist == true)
        {
            argv2 = filename = argv[2];
            if(filename != NULL)
            {
                //delete old motion if any
                if (pSampledMotion != NULL)
                {
                    delete pSampledMotion;
                    pSampledMotion = NULL;
                }
                if (pInterpMotion != NULL)
                {
```

```
        delete pInterpMotion;
        pInterpMotion = NULL;
    }

    //Read motion (.amc) file and create a motion
    pSampledMotion = new Motion(filename, MOCAP_SCALE, pActor);

    //set sampled motion for display
    displayer.loadMotion(pSampledMotion);

    //Tell actor to perform the first pose ( first posture )
    pActor->setPosture(displayer.m_pMotion[0]->m_pPostures[0]);

    frame_slider->maximum((double)displayer.m_pMotion[0]->m_NumFrames );

    nFrameNum=0;
}
}
else
    printf("Load Actor first.\n");
    nFrameInc=4;          // Current frame and frame increment
Play=ON;                // Some Flags for player
Repeat=OFF;
Record=ON;
Background=OFF;
Light=OFF; // Flags indicating if the object exists
Record_filename = ""; // Recording file name
recmode=1;
}
    glwindow->redraw();
}
Fl::add_idle(idle);
return Fl::run();}
```

```
//-----
// Filename: Quaternion.h
// Description:
// quaternion class header file
//-----
```

```
#ifndef _QUATERNION_H_
#define _QUATERNION_H_
#include <iostream>
#include "Vector3.h"
using namespace std;
```

```
class Quaternion{
```

```
protected:
```

```
    double w;
    double i;
    double j;
    double k;
```

```
public:
```

```
    //constructor: identity quaternion
```

```
    Quaternion();
    Quaternion(double s,double m,double n,double o);
    Quaternion(double s,Vector3 v);
```

```
    //destructor
    ~Quaternion();
```

```
    //get
    double getW();
    double getI();
    double getJ();
    double getK();
```

lyit | Institiúid Teicneolaíochta Leitir Ceanaínn
Letterkenny Institute of Technology

```

void setQuaternion(double s,double m,double n,double o);
//Sets all four components of a quaternion.

    void setRotationAboutX(double x);
//Sets a quaternion to represent a rotation about the x-axis.
    void setRotationAboutY(double y) ;
//Sets a quaternion to represent a rotation about the y-axis.
    void setRotationAboutZ(double z);
//Sets a quaternion to represent a rotation about the z-axis.
    void setRotationAboutAxis(double a);
//Sets a quaternion to represent a rotation about a given axis.

    void getRotationMatrix();
//Converts a quaternion to a 3 x 3 matrix.
    void setRotationMatrix() ;
//Converts a 3 x 3 matrix to a quaternion.

Quaternion & operator =(const Quaternion & q);
//assignment

Quaternion & operator +=(const Quaternion & q);
Quaternion & operator -=(const Quaternion & q);
Quaternion & operator *=(const Quaternion & q);
Quaternion & operator *=(const double & f);
Quaternion & operator /=(const Quaternion & q);
Quaternion & operator /=(const double & f);

Quaternion operator +(const Quaternion & q) const;
Quaternion operator -(const Quaternion & q) const;
Quaternion operator *(const Quaternion & q) const;
//calculate cross product with quaternions
Quaternion operator *(const double & f) const;
//calculate produnt quaternion with a scalar
Quaternion operator /(const Quaternion & q) const;
//calculate the inverse of Quaternion
Quaternion operator /(const double & f) const;

```



```
//calculate the inverse of Quaternion

    bool operator ==(const Quaternion& q) const;
//equality between two quaternions
    bool operator !=(const Quaternion& q) const;

    double magnitude() const;
//calculate a magnitude of a quaternion.

    Quaternion conjugate() const;
//negate vector portion of quaternion
    Quaternion Inverse () const;
    Quaternion Exp () const;
    Quaternion Log () const;
    Quaternion Pow(const Quaternion &q, double exponent);

    friend ostream & operator<< (ostream &os, const Quaternion & q);
};

//Linear interpolation-'lerp'
extern Quaternion lerp(const Quaternion & q1, const Quaternion &q2, double t );
extern Quaternion Slerp(const Quaternion & q1, const Quaternion &q2, double t );

#endif
```

Quaternion.cpp

```
//-----  
// Filename: Quaternion.cpp  
// Description:  
// quaternion class header file  
//-----  
#include "Quaternion.h"  
#include <cmath>  
  
//set default constructor to identity quaternion  
Quaternion::Quaternion()  
{  
    w = 1.0f;  
    i = 0.0f;  
    j = 0.0f;  
    k = 0.0f;  
}  
  
Quaternion::Quaternion(double s, double m, double n, double r)  
{  
    w = s;  
    i = m;  
    j = n;  
    k = r;  
}  
  
//get methods  
double Quaternion::getW()  
{  
    return w;  
}  
  
double Quaternion::getI()  
{  
    return i;  
}  
  
double Quaternion::getJ()  
{
```

a, double o)



```
    return j;
}
```

```
double Quaternion::getK()
{
    return k;
}
```

```
//Sets all four components of a quaternion.
```

```
void Quaternion::setQuaternion(double s,double m,double n,double o)
{
    this->w = s;
    this->i = m;
    this->j = n;
    this->k = o;
}
```

```
Quaternion & Quaternion::operator =(const Quaternion &q)
{
    this->w = q.w;
    this->i = q.i;
    this->j = q.j;
    this->k = q.k;
    return *this;
}
```

```
Quaternion & Quaternion::operator +=(const Quaternion &q)
{
    *this = *this + q;
    return *this;
}
```

```
Quaternion & Quaternion::operator -=(const Quaternion &q)
{
    *this = *this - q;
    return *this;
}
```

```
Quaternion & Quaternion::operator *=(const Quaternion &q)
{
    *this = *this * q;
    return *this;
}
Quaternion & Quaternion::operator /=(const Quaternion &q)
{
    *this = *this / q;
    return *this;
}
Quaternion & Quaternion::operator /=(const double &f)
{
    *this = *this / f;
    return *this;
}

Quaternion Quaternion::operator +(const Quaternion &q) const
{
    Quaternion result;
    result.w = w + q.w;
    result.i = i + q.i;
    result.j = j + q.j;
    result.k = k + q.k;

    return result;
}

Quaternion Quaternion::operator -(const Quaternion &q) const
{
    Quaternion result;
    result.w = w - q.w;
    result.i = i - q.i;
    result.j = j - q.j;
    result.k = k - q.k;

    return result;
}
```

```

//calculate cross product with quaternions
Quaternion Quaternion::operator *(const Quaternion &q) const
{
    Quaternion result;

    result.w = w*q.w - i*q.i - j*q.j - k*q.k;
    result.i = w*q.i + i*q.w + k*q.j - j*q.k;
    result.j = w*q.j + j*q.w + i*q.k - k*q.i;
    result.k = w*q.k + k*q.w + j*q.i - i*q.j;

    return result;
}

Quaternion Quaternion::operator *(const double &f) const
{
    Quaternion result;
    result.w = this->w * f;
    result.i = this->i * f;
    result.j = this->j * f;
    result.k = this->k * f;
    return result;
}

Quaternion Quaternion::operator /(const double &f) const
{
    Quaternion result;
    result.w = this->w / f;
    result.i = this->i / f;
    result.j = this->j / f;
    result.k = this->k / f;
    return result;
}

ostream & operator<< (ostream &os, const Quaternion &q)
{
    os << "( " << q.w<< " " << q.i<< " " << q.j<< " " << q.k<< " )";
}

```

```

    return os;
}

bool Quaternion::operator ==(const Quaternion &q) const
{
    return (w==q.w && i==q.i && j==q.j && k==q.k) ? true : false;
}

bool Quaternion::operator !=(const Quaternion &q) const
{
    return (w!=q.w || i!=q.i || j!=q.j || k!=q.k) ? true : false;
}

//Normalizes a quaternion.
double Quaternion::magnitude() const
{
    return sqrt( w*w + i*i + j*j + k*k );
}

//Conjugate
Quaternion Quaternion::conjugate()const
{
    return Quaternion( w, -i, -j, -k);
}

Quaternion Quaternion::Inverse() const
{
    return (*this).conjugate()/(*this).magnitude();
}

Quaternion Quaternion::Log() const
{
    // Check for the case of an identity quaternion.
    // prevent divide by zero

    if ( fabs(this->w) > .9999f) {
        return *this;
    }
}

```

```

    // Extract the half angle alpha (alpha = theta/2)
    double a = acos(this->w);
    return Quaternion (0,a*i,a*j,a*k);
}

Quaternion Quaternion::Pow(const Quaternion &q, double exponent)
{
    // Check for the case of an identity quaternion.
    // This will protect against divide by zero

    if (fabs(q.w) > .9999f) {
        return q;
    }
    // Extract the half angle alpha (alpha = theta/2)
    double alpha = acos(q.w);

    // Compute new alpha value
    double newAlpha = alpha * exponent;

    // Compute new w value
    Quaternion result;
    result.w = cos(newAlpha);

    // Compute new xyz values

    double mult = sin(newAlpha) / sin(alpha);
    result.i = q.i * mult;
    result.j = q.j * mult;
    result.k = q.k * mult;
    return result;
}

Quaternion Quaternion::Exp () const
{
    // Check for the case of an identity quaternion.
    // prevent divide by zero
    if (this->w!=0){
        cout<< "input should be Vector type Quaternion!!"<<endl;
    }
}

```



```

        return *this;
    }
    else
    {
        double mag;
        mag = this->magnitude();
        if (mag==0)
        {
            cout<<"cannot divide by zero";
            return *this;
        }
        else
        {
            double mult = 0.0f;
            mult = sin( mag ) / mag;
            cout<<mult<<endl;

            return Quaternion( cos( mag ), mult*i,mult*j, mult*k);
        }
    }
}

Quaternion lerp(const Quaternion &q1, const Quaternion &q2, double t )
{
    //((1-t)q1+ tq2)/|((1-t)q1+ tq2)|
    Quaternion tmp;
    tmp = q1 - q1*t + q2*t;
    tmp = tmp / tmp.magnitude();

    return tmp;
}

```

Vector.cpp

```
//-----  
// Filename: Vector.cpp  
// Description:  
//  
//-----  
  
#include <math.h>  
#include <stdio.h>  
#include "transform.h"  
#include "types.h"  
  
#include "vector.h"  
  
//#include "mathclass.h"  
  
vector operator-( vector const& a, vector const& b )  
{  
    vector c;  
  
    c.p[0] = a.p[0] - b.p[0];  
    c.p[1] = a.p[1] - b.p[1];  
    c.p[2] = a.p[2] - b.p[2];  
  
    return c;  
}  
  
vector operator+( vector const& a, vector const& b )  
{  
    vector c;  
  
    c.p[0] = a.p[0] + b.p[0];  
    c.p[1] = a.p[1] + b.p[1];  
    c.p[2] = a.p[2] + b.p[2];  
  
    return c;  
}
```

lyit

Institiúid Teicneolaíochta Leitir Ceanáin
Letterkenny Institute of Technology

```
vector operator/( vector const& a, float b )
{
    vector c;

    c.p[0] = a.p[0] / b;
    c.p[1] = a.p[1] / b;
    c.p[2] = a.p[2] / b;

    return c;
}

//multip
vector operator*( vector const& a, float b )
{
    vector c;

    c.p[0] = a.p[0] * b;
    c.p[1] = a.p[1] * b;
    c.p[2] = a.p[2] * b;

    return c;
}

//cross product
vector operator*( vector const& a, vector const& b )
{
    vector c;

    c.p[0] = a.p[1]*b.p[2] - a.p[2]*b.p[1];
    c.p[1] = a.p[2]*b.p[0] - a.p[0]*b.p[2];
    c.p[2] = a.p[0]*b.p[1] - a.p[1]*b.p[0];

    return c;
}

//dot product
float operator%( vector const& a, vector const& b )
```

lyit | Institiúid Teicneolaíochta Leitir Ceanáin
Letterkenny Institute of Technology

```
{
    return ( a.p[0]*b.p[0] + a.p[1]*b.p[1] + a.p[2]*b.p[2] );
}

vector interpolate( float t, vector const& a, vector const& b )
{
    return a*(1.0-t) + b*t;
}

float len( vector const& v )
{
    return sqrt( v.p[0]*v.p[0] + v.p[1]*v.p[1] + v.p[2]*v.p[2] );
}

float
vector::length() const
{
    return sqrt( p[0]*p[0] + p[1]*p[1] + p[2]*p[2] );
}

float angle( vector const& a, vector const& b )
{
    return acos( (a*b)/(len(a)*len(b)) );
}

//-----
vector normalize( vector const& v )
{
    vector vec_normed;
    double norm ;
    norm = len(v);
    if ((norm) < 1e-6)
    {
        vec_normed.setValue(1.0,0,0);

        //fprintf(stderr,"Warning: a zero vector was given to VecNormalize\n") ;
    }
}
```

```
        return vec_normed;
    }
    norm = 1.0 / norm ;
    vec_normed= v * norm ;

    return vec_normed;
}
```

```
float vector::dot(const vector &a) const {
    float result;
    result = p[0]*a.p[0] + p[1]*a.p[1] + p[2]*a.p[2];
    return result;
}
```

```
vector vector::cross(const vector &a) const {

    vector result;

    result.p[0] = p[1]*a.p[2] - p[2]*a.p[1];
    result.p[1] = -p[0]*a.p[2] + p[2]*a.p[0];
    result.p[2] = p[0]*a.p[1] - p[1]*a.p[0];

    return result;
}
```

```
void VecCrossProd(vector c, const vector a, const vector b)
{
    c.p[0] = a.p[1]*b.p[2] - a.p[2]*b.p[1] ;
    c.p[1] = -a.p[0]*b.p[2] + a.p[2]*b.p[0] ;
    c.p[2] = a.p[0]*b.p[1] - a.p[1]*b.p[0] ;
}
```

```
void
VecSubtract(vector c, const vector a, const vector b)
{
```

```
    c.p[0] = a.p[0] - b.p[0] ;
    c.p[1] = a.p[1] - b.p[1] ;
    c.p[2] = a.p[2] - b.p[2] ;
}

void
VecAdd(vector c, const vector a, const vector b)
{
    c.p[0] = a.p[0] + b.p[0] ;
    c.p[1] = a.p[1] + b.p[1] ;
    c.p[2] = a.p[2] + b.p[2] ;
}

void
VecCopy(vector c, const vector a)
{
    c.p[0] = a.p[0] ;
    c.p[1] = a.p[1] ;
    c.p[2] = a.p[2] ;
}

void
VecSwap(vector a, vector b)
{
    vector temp;
    VecCopy(temp, a);
    VecCopy(a, b);
    VecCopy(b, temp);
}

double VecDotProd(const vector a, const vector b)
{
    return( a.p[0]*b.p[0]+a.p[1]*b.p[1]+a.p[2]*b.p[2] ) ;
}
```



```
void VecNumMul(vector c, const vector a, float n)
{
    c.p[0] = a.p[0]*n ;
    c.p[1] = a.p[1]*n ;
    c.p[2] = a.p[2]*n ;
}
double Distance(const vector c, const vector a)
{
    double dist;
    double sqx = (a.p[0]-c.p[0])*(a.p[0]-c.p[0]);
    double sqy = (a.p[1]-c.p[1])*(a.p[1]-c.p[1]);
    double sqz = (a.p[2]-c.p[2])*(a.p[2]-c.p[2]);

    return sqrt(sqx+sqy+sqz);
}

vector CatmullRom(const vector v1,const vector v2,const vector v3,const vector v4,float t)
{
    float t2 = t * t;
    float t3 = t2 * t;
    vector out ;

    out.p[0]= 0.5f * ( ( 2.0f * v2.p[0] ) +( -v1.p[0] + v3.p[0] ) * t +( 2.0f * v1.p[0] - 5.0f * v2.p[0] + 4 * v3.p[0]
- v4.p[0] ) * t2 +( -v1.p[0] + 3.0f * v2.p[0] - 3.0f * v3.p[0] + v4.p[0] ) * t3 );
    out.p[1]= 0.5f * ( ( 2.0f * v2.p[1] ) +( -v1.p[1] + v3.p[1] ) * t +( 2.0f * v1.p[1] - 5.0f * v2.p[1] + 4 * v3.p[1]
- v4.p[1] ) * t2 +( -v1.p[1] + 3.0f * v2.p[1] - 3.0f * v3.p[1] + v4.p[1] ) * t3 );
    out.p[2]= 0.5f * ( ( 2.0f * v2.p[2] ) +( -v1.p[2] + v3.p[2] ) * t +( 2.0f * v1.p[2] - 5.0f * v2.p[2] + 4 * v3.p[2]
- v4.p[2] ) * t2 +( -v1.p[2] + 3.0f * v2.p[2] - 3.0f * v3.p[2] + v4.p[2] ) * t3 );

    return out;
}
```

Quaternion.h

```
//-----  
// Filename: Quaternion.h  
// Description:  
// quaternion class header file  
//-----  
  
#ifndef _QUATERNION_H_  
#define _QUATERNION_H_  
#include <iostream>  
#include "Vector3.h"  
using namespace std;  
  
class Quaternion{  
  
protected:  
    double w;  
    double i;  
    double j;  
    double k;  
public:  
    //constructor: identity quaternion  
    Quaternion();  
    Quaternion(double s,double m,double n,double o);  
    Quaternion(double s,Vector3 v);  
  
    //destructor  
    ~Quaternion();  
  
    //get  
    double getW();  
    double getI();  
    double getJ();  
    double getK();
```



```

void setQuaternion(double s,double m,double n,double o);
//Sets all four components of a quaternion.

    void setRotationAboutX(double x);
//Sets a quaternion to represent a rotation about the x-axis.
    void setRotationAboutY(double y) ;
//Sets a quaternion to represent a rotation about the y-axis.
    void setRotationAboutZ(double z);
//Sets a quaternion to represent a rotation about the z-axis.
    void setRotationAboutAxis(double a);
//Sets a quaternion to represent a rotation about a given axis.

    void getRotationMatrix();
//Converts a quaternion to a 3 x 3 matrix.
    void setRotationMatrix() ;
//Converts a 3 x 3 matrix to a quaternion.

Quaternion & operator =(const Quaternion & q);
//assignment

Quaternion & operator +=(const Quaternion & q);
Quaternion & operator -=(const Quaternion & q);
Quaternion & operator *=(const Quaternion & q);
Quaternion & operator *=(const double & f);
Quaternion & operator /=(const Quaternion & q);
Quaternion & operator /=(const double & f);

Quaternion operator +(const Quaternion & q) const;
Quaternion operator -(const Quaternion & q) const;
Quaternion operator *(const Quaternion & q) const;
//calculate cross product with quaternions
Quaternion operator *(const double & f) const;
//calculate product quaternion with a scalar
Quaternion operator /(const Quaternion & q) const;
//calculate the inverse of Quaternion
Quaternion operator /(const double & f) const;

```

```
//calculate the inverse of Quaternion

    bool operator ==(const Quaternion& q) const;
//equality between two quaternions
    bool operator !=(const Quaternion& q) const;

    double magnitude() const;
//calculate a magnitude of a quaternion.

    Quaternion conjugate() const;
//negate vector portion of quaternion
    Quaternion Inverse () const;
    Quaternion Exp () const;
    Quaternion Log () const;
    Quaternion Pow(const Quaternion &q, double exponent);

    friend ostream & operator<< (ostream &os, const Quaternion & q);
};

//Linear interpolation-'lerp'
extern Quaternion lerp(const Quaternion &q1, const Quaternion &q2, double t );
extern Quaternion Slerp(const Quaternion &q1, const Quaternion &q2, double t );

#endif
```

Quaternion.cpp

```
//-----  
// Filename: Quaternion.cpp  
// Description:  
// quaternion class header file  
//-----  
#include "Quaternion.h"  
#include <cmath>  
  
//set default constructor to identity quaternion  
Quaternion::Quaternion()  
{  
    w = 1.0f;  
    i = 0.0f;  
    j = 0.0f;  
    k = 0.0f;  
}  
  
Quaternion::Quaternion(double s, double m, double n, double o)  
{  
    w = s;  
    i = m;  
    j = n;  
    k = o;  
}  
  
//get methods  
double Quaternion::getW()  
{  
    return w;  
}  
  
double Quaternion::getI()  
{  
    return i;  
}  
  
double Quaternion::getJ()  
{
```

a, double o)

lyit | Institiúid Teicneolaíochta Leitir Ceannainn
Letterkenny Institute of Technology

```
    return j;
}
```

```
double Quaternion::getK()
{
    return k;
}
```

```
//Sets all four components of a quaternion.
```

```
void Quaternion::setQuaternion(double s,double m,double n,double o)
{
    this->w = s;
    this->i = m;
    this->j = n;
    this->k = o;
}
```

```
Quaternion & Quaternion::operator =(const Quaternion &q)
{
    this->w = q.w;
    this->i = q.i;
    this->j = q.j;
    this->k = q.k;
    return *this;
}
```

```
Quaternion & Quaternion::operator +=(const Quaternion &q)
{
    *this = *this + q;
    return *this;
}
```

```
Quaternion & Quaternion::operator -=(const Quaternion &q)
{
    *this = *this - q;
    return *this;
}
```



```
Quaternion & Quaternion::operator *=(const Quaternion &q)
{
    *this = *this * q;
    return *this;
}
Quaternion & Quaternion::operator /=(const Quaternion &q)
{
    *this = *this / q;
    return *this;
}
Quaternion & Quaternion::operator /=(const double &f)
{
    *this = *this / f;
    return *this;
}

Quaternion Quaternion::operator +(const Quaternion &q) const
{
    Quaternion result;
    result.w = w + q.w;
    result.i = i + q.i;
    result.j = j + q.j;
    result.k = k + q.k;

    return result;
}

Quaternion Quaternion::operator -(const Quaternion &q) const
{
    Quaternion result;
    result.w = w - q.w;
    result.i = i - q.i;
    result.j = j - q.j;
    result.k = k - q.k;

    return result;
}
```

```

//calculate cross product with quaternions
Quaternion Quaternion::operator *(const Quaternion &q) const
{
    Quaternion result;

    result.w = w*q.w - i*q.i - j*q.j - k*q.k;
    result.i = w*q.i + i*q.w + k*q.j - j*q.k;
    result.j = w*q.j + j*q.w + i*q.k - k*q.i;
    result.k = w*q.k + k*q.w + j*q.i - i*q.j;

    return result;
}

Quaternion Quaternion::operator *(const double &f) const
{
    Quaternion result;
    result.w = this->w * f;
    result.i = this->i * f;
    result.j = this->j * f;
    result.k = this->k * f;
    return result;
}

Quaternion Quaternion::operator /(const double & f) const
{
    Quaternion result;
    result.w = this->w / f;
    result.i = this->i / f;
    result.j = this->j / f;
    result.k = this->k / f;
    return result;
}

ostream & operator<< (ostream &os, const Quaternion &q)
{
    os << "( " << q.w<< " " << q.i<< " " << q.j<< " " << q.k<< " )";
}

```

```

    return os;
}

bool Quaternion::operator ==(const Quaternion &q) const
{
    return (w==q.w && i==q.i && j==q.j && k==q.k) ? true : false;
}

bool Quaternion::operator !=(const Quaternion &q) const
{
    return (w!=q.w || i!=q.i || j!=q.j || k!=q.k) ? true : false;
}

//Normalizes a quaternion.
double Quaternion::magnitude() const
{
    return sqrt( w*w + i*i + j*j + k*k );
}

//Conjugate
Quaternion Quaternion::conjugate()const
{
    return Quaternion( w, -i, -j, -k);
}

Quaternion Quaternion::Inverse() const
{
    return (*this).conjugate()/(*this).magnitude();
}

Quaternion Quaternion::Log() const
{
    // Check for the case of an identity quaternion.
    // prevent divide by zero

    if ( fabs(this->w) > .9999f) {
        return *this;
    }
}

```

```

    // Extract the half angle alpha (alpha = theta/2)
    double a = acos(this->w);
    return Quaternion (0,a*i,a*j,a*k);
}

Quaternion Quaternion::Pow(const Quaternion &q, double exponent)
{
    // Check for the case of an identity quaternion.
    // This will protect against divide by zero

    if (fabs(q.w) > .9999f) {
        return q;
    }
    // Extract the half angle alpha (alpha = theta/2)
    double alpha = acos(q.w);

    // Compute new alpha value
    double newAlpha = alpha * exponent;

    // Compute new w value
    Quaternion result;
    result.w = cos(newAlpha);

    // Compute new xyz values

    double mult = sin(newAlpha) / sin(alpha);
    result.i = q.i * mult;
    result.j = q.j * mult;
    result.k = q.k * mult;
    return result;
}

Quaternion Quaternion::Exp () const
{
    // Check for the case of an identity quaternion.
    // prevent divide by zero
    if (this->w!=0){
        cout<< "input should be Vector type Quaternion!!"<<endl;
    }
}

```

```

        return *this;
    }
    else
    {
        double mag;
        mag = this->magnitude();
        if (mag==0)
        {
            cout<<"cannot divide by zero";
            return *this;
        }
        else
        {
            double mult = 0.0f;
            mult = sin( mag ) / mag;
            cout<<mult<<endl;

            return Quaternion( cos( mag ), mult*i,mult*j, mult*k);
        }
    }
}

Quaternion lerp(const Quaternion &q1, const Quaternion &q2, double t )
{
    //((1-t)q1+ tq2) / ||((1-t)q1+ tq2)||
    Quaternion tmp;
    tmp = q1 - q1*t + q2*t;
    tmp = tmp / tmp.magnitude();

    return tmp;
}

```

Vector.cpp

```
//-----  
// Filename: Vector.cpp  
// Description:  
//  
//-----  
  
#include <math.h>  
#include <stdio.h>  
#include "transform.h"  
#include "types.h"  
  
#include "vector.h"  
  
//#include "mathclass.h"  
  
vector operator-( vector const& a, vector const& b )  
{  
    vector c;  
  
    c.p[0] = a.p[0] - b.p[0];  
    c.p[1] = a.p[1] - b.p[1];  
    c.p[2] = a.p[2] - b.p[2];  
  
    return c;  
}  
  
vector operator+( vector const& a, vector const& b )  
{  
    vector c;  
  
    c.p[0] = a.p[0] + b.p[0];  
    c.p[1] = a.p[1] + b.p[1];  
    c.p[2] = a.p[2] + b.p[2];  
  
    return c;  
}
```

lyit | **Institiúid Telcneolaíochta Leitir Ceanáin**
Letterkenny Institute of Technology

```
vector operator/( vector const& a, float b )
{
    vector c;

    c.p[0] = a.p[0] / b;
    c.p[1] = a.p[1] / b;
    c.p[2] = a.p[2] / b;

    return c;
}

//multip
vector operator*( vector const& a, float b )
{
    vector c;

    c.p[0] = a.p[0] * b;
    c.p[1] = a.p[1] * b;
    c.p[2] = a.p[2] * b;

    return c;
}

//cross product
vector operator*( vector const& a, vector const& b )
{
    vector c;

    c.p[0] = a.p[1]*b.p[2] - a.p[2]*b.p[1];
    c.p[1] = a.p[2]*b.p[0] - a.p[0]*b.p[2];
    c.p[2] = a.p[0]*b.p[1] - a.p[1]*b.p[0];

    return c;
}

//dot product
float operator%( vector const& a, vector const& b )
```



```
{
    return ( a.p[0]*b.p[0] + a.p[1]*b.p[1] + a.p[2]*b.p[2] );
}

vector interpolate( float t, vector const& a, vector const& b )
{
    return a*(1.0-t) + b*t;
}

float len( vector const& v )
{
    return sqrt( v.p[0]*v.p[0] + v.p[1]*v.p[1] + v.p[2]*v.p[2] );
}

float
vector::length() const
{
    return sqrt( p[0]*p[0] + p[1]*p[1] + p[2]*p[2] );
}

float angle( vector const& a, vector const& b )
{
    return acos( (a*b)/(len(a)*len(b)) );
}

//-----
vector normalize( vector const& v )
{
    vector vec_normed;
    double norm ;
    norm = len(v);
    if ((norm) < 1e-6)
    {
        vec_normed.setValue(1.0,0,0);

        //fprintf(stderr,"Warning: a zero vector was given to VecNormalize\n");
    }
}
```

```
        return vec_normed;
    }
    norm = 1.0 / norm ;
    vec_normed= v * norm ;

    return vec_normed;
}

float vector::dot(const vector &a) const {
    float result;
    result = p[0]*a.p[0] + p[1]*a.p[1] + p[2]*a.p[2];
    return result;
}

vector vector::cross(const vector &a) const {

    vector result;

    result.p[0] = p[1]*a.p[2] - p[2]*a.p[1];
    result.p[1] = -p[0]*a.p[2] + p[2]*a.p[0];
    result.p[2] = p[0]*a.p[1] - p[1]*a.p[0];

    return result;
}

void VecCrossProd(vector c, const vector a, const vector b)
{
    c.p[0] = a.p[1]*b.p[2] - a.p[2]*b.p[1] ;
    c.p[1] = -a.p[0]*b.p[2] + a.p[2]*b.p[0] ;
    c.p[2] = a.p[0]*b.p[1] - a.p[1]*b.p[0] ;
}

void
VecSubtract(vector c, const vector a, const vector b)
{
```

```
    c.p[0] = a.p[0] - b.p[0] ;
    c.p[1] = a.p[1] - b.p[1] ;
    c.p[2] = a.p[2] - b.p[2] ;
}

void
VecAdd(vector c, const vector a, const vector b)
{
    c.p[0] = a.p[0] + b.p[0] ;
    c.p[1] = a.p[1] + b.p[1] ;
    c.p[2] = a.p[2] + b.p[2] ;
}

void
VecCopy(vector c, const vector a)
{
    c.p[0] = a.p[0] ;
    c.p[1] = a.p[1] ;
    c.p[2] = a.p[2] ;
}

void
VecSwap(vector a, vector b)
{
    vector temp;
    VecCopy(temp, a);
    VecCopy(a, b);
    VecCopy(b, temp);
}

double VecDotProd(const vector a, const vector b)
{
    return( a.p[0]*b.p[0]+a.p[1]*b.p[1]+a.p[2]*b.p[2] ) ;
}
```

```
void VecNumMul(vector c, const vector a, float n)
{
    c.p[0] = a.p[0]*n ;
    c.p[1] = a.p[1]*n ;
    c.p[2] = a.p[2]*n ;
}
double Distance(const vector c, const vector a)
{
    double dist;
    double sqx = (a.p[0]-c.p[0])*(a.p[0]-c.p[0]);
    double sqy = (a.p[1]-c.p[1])*(a.p[1]-c.p[1]);
    double sqz = (a.p[2]-c.p[2])*(a.p[2]-c.p[2]);

    return sqrt(sqx+sqy+sqz);
}

vector CatmullRom(const vector v1,const vector v2,const vector v3,const vector v4,float t)
{
    float t2 = t * t;
    float t3 = t2 * t;
    vector out ;

    out.p[0]= 0.5f * ( ( 2.0f * v2.p[0] ) +( -v1.p[0] + v3.p[0] ) * t +( 2.0f * v1.p[0] - 5.0f * v2.p[0] + 4 * v3.p[0]
- v4.p[0] ) * t2 +( -v1.p[0] + 3.0f * v2.p[0] - 3.0f * v3.p[0] + v4.p[0] ) * t3 );
    out.p[1]= 0.5f * ( ( 2.0f * v2.p[1] ) +( -v1.p[1] + v3.p[1] ) * t +( 2.0f * v1.p[1] - 5.0f * v2.p[1] + 4 * v3.p[1]
- v4.p[1] ) * t2 +( -v1.p[1] + 3.0f * v2.p[1] - 3.0f * v3.p[1] + v4.p[1] ) * t3 );
    out.p[2]= 0.5f * ( ( 2.0f * v2.p[2] ) +( -v1.p[2] + v3.p[2] ) * t +( 2.0f * v1.p[2] - 5.0f * v2.p[2] + 4 * v3.p[2]
- v4.p[2] ) * t2 +( -v1.p[2] + 3.0f * v2.p[2] - 3.0f * v3.p[2] + v4.p[2] ) * t3 );

    return out;
}
```