

# Skeleton-based Rigid Skinning for Character Animation

Η ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

υποβάλλεται στην  
ορισθείσα από την Γενική Συνέλευση Ειδικής Σύθεσης  
του Τμήματος Πληροφορικής Εξεταστική Επιτροπή

από τον

Ανδρέα-Αλέξανδρο Βασιλάκη

ως μέρος των Υποχρεώσεων για τη λήψη του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ  
ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ ΣΤΟ ΛΟΓΙΣΜΙΚΟ

Ιούλιος 2008



# DEDICATION

---

To the most wonderful girl on earth

# ACKNOWLEDGEMENTS

---

This work could not be accomplished without support from many people.

First of all, I am extremely thankful to my supervisor Assistant Professor Ioannis Fudos for his valuable help, his leading advices and his inexhaustible patience that he has shown during the elaboration of this thesis. Throughout my master work he encouraged me to pursue my own research interests and to develop independent thinking and research skills.

Furthermore, I need to thank my colleagues and best friends G. Karpathios, V. Lappas and M. Markoulakis for their honest and moral support through this challenging journey.

Finally, I would not forget my close friend PhD candidate A. Kalogeratos for the endless, pleasure and useful dialogues we had for the past two years.

I consider myself very lucky to have so many great people in my life.



# TABLE OF CONTENTS

---

<b>1</b>	<b>INTRODUCTION</b>	<b>1</b>
1.1	Definition of the Problem . . . . .	3
1.2	Applications . . . . .	5
1.3	Related Work . . . . .	6
1.3.1	Skeletonization . . . . .	6
1.3.2	Skinning . . . . .	7
1.4	Structure of Thesis . . . . .	8
<b>2</b>	<b>BACKGROUND MATERIAL</b>	<b>9</b>
2.1	Vector and Tools . . . . .	9
2.1.1	Vector . . . . .	9
2.1.2	2D Implicit Curves . . . . .	14
2.1.3	2D Parametric Curves . . . . .	14
2.1.4	3D Implicit Surfaces . . . . .	18
2.1.5	3D Parametric Surfaces . . . . .	19
2.2	Transformations . . . . .	21
2.2.1	3D Matrix Theory . . . . .	22
2.2.2	Quaternion Theory . . . . .	27
2.2.3	Performance Issues . . . . .	31
2.3	Computational Geometry Techniques . . . . .	32
2.3.1	Kernel's Centroid . . . . .	33
2.3.2	Minimum Bounding Volume . . . . .	34
2.4	Data Structures for geometric objects . . . . .	37
<b>3</b>	<b>SKELETONIZATION</b>	<b>39</b>
3.1	Skeleton Representation . . . . .	40
3.2	Skeleton Extraction . . . . .	41
3.2.1	Opening Method . . . . .	41
3.2.2	Centroid Method . . . . .	42
3.2.3	Principal Axis Method . . . . .	43
3.3	Skeleton Refinement . . . . .	49
3.3.1	Local Refinement Method . . . . .	50
3.3.2	Parent Refinement Method . . . . .	51

<b>4</b>	<b>ARTICULATED ANIMATION</b>	<b>54</b>
4.1	Representing Articulated Figures . . . . .	55
4.2	Keyframing Animation System . . . . .	55
4.3	Forward Kinematics Skinning System . . . . .	61
4.4	Gap Reconstruction Process . . . . .	65
4.4.1	Remove Vertices . . . . .	66
4.4.2	Add New Vertices . . . . .	68
4.4.3	Triangulate Vertices . . . . .	70
4.4.4	Compute Normal Vectors . . . . .	71
<b>5</b>	<b>IMPLEMENTATION AND RESULTS</b>	<b>73</b>
5.1	Implementation Details . . . . .	73
5.2	Experiment Results . . . . .	75
5.2.1	Skeletonization results . . . . .	75
5.2.2	Animation results . . . . .	89
<b>6</b>	<b>Conclusions and Future Work</b>	<b>99</b>

# LIST OF FIGURES

---

1.1	Toy Story is a 1995 Pixar Animation Studios [107] film using Computer Animation. . . . .	2
1.2	Illustration of skeleton subspace deformation algorithm. The deformed position of a point $V$ lies on the line $V_1V_2$ defined by the images of that point rigidly transformed by the neighboring skeletal coordinate frames, resulting in the characteristic “joint collapsing” problem. . . . .	8
2.1	A vector going from A to B. . . . .	10
2.2	Coordinate systems. Part(a) The two-dimensional coordinate system. Part(b) The three-dimensional coordinate systems: The left-handed orientation is depicted on the left, and the right-handed is depicted on the right. . . . .	10
2.3	Vector addition: Parallelogram rule. . . . .	11
2.4	Vector scaling. . . . .	11
2.5	Vector subtracting. . . . .	11
2.6	Normalizing a vector. . . . .	12
2.7	The dot product of two vectors. . . . .	13
2.8	Scalar projection of a vector $b$ in the direction of a vector $a$ . . . . .	13
2.9	The cross product of two vectors. . . . .	13
2.10	Three implicit lines. . . . .	14
2.11	Line-Line intersection. . . . .	15
2.12	Hermite Curve. . . . .	16
2.13	Cubic Bezier Curve. . . . .	17
2.14	Point-plane distance and projection of point $P_1$ on the plane. . . . .	19
2.15	The dihedral angle between two planes. . . . .	19
2.16	Circle. . . . .	20
2.17	Normal of a triangle. . . . .	21
2.18	Barycentric coordinates. . . . .	21
2.19	Illustrating the loss of one degree of freedom (Gimbal Lock). Part(a) x-roll $\theta_1$ . Part(b) x-roll $\theta_1$ followed by y-roll $\pi/2$ . x-axis effectively gets rotated to $x'$ -axis. Part(c) followed by z-roll $\theta_3$ . z-roll $\theta_3$ same as x-roll $-\theta_1$ . . . . .	24
2.20	Rotating vector $P$ about an arbitrary point $V$ . . . . .	25
2.21	Alignment of vector with a unit vector $k$ which lies on positive z-axis . . . . .	26
2.22	Angular displacement $(\theta, n)$ of $r$ . . . . .	27

2.23	Kernel of a polygon is shown at this figure in red. . . . .	34
2.24	In a sphere-sphere intersection, the routine may report that collision has occurred when it really hasn't. . . . .	35
2.25	Objects and their axis-aligned box boundaries. . . . .	35
2.26	Successive AABBs for a spinning rod (as viewed from the side). . . . .	36
2.27	Separating axis (intervals do not overlap). . . . .	36
2.28	Convex hull of a set of 2D points. . . . .	37
2.29	Example of a triangle mesh representing a dolphin. . . . .	38
3.1	Skeleton representation file format. . . . .	41
3.2	The produced skeleton hierarchy for Figure 3.1 example. . . . .	41
3.3	The generated opening centroids between P1 and the connecting neighbor components P2 and P3. . . . .	42
3.4	(a) A simple example of a skeletal representation using the Opening Method. (b) An example that illustrates the inappropriate skeleton representation that arises when skeletonization is based only on opening centroids. . . . .	42
3.5	(a) A simple example of a skeletal representation using the Centroid Method. (b) The Opening Method problem in 3.4b addressed using the Centroid Method. . . . .	43
3.6	(a) An example that illustrates the problem that arises when skeletonization is based only on centroids. (b) This problem can be solved using the principal axis. Points $b$ and $d$ are the centers of the openings and $a$ , $c$ and $e$ the kernel's centers of the components $P1$ , $P2$ and $P3$ , respectively. . . . .	44
3.7	Difference between $PA_{MC}(CH)$ and $PA_{KC}(C)$ . The projection of opening centroid $oc_2$ ( $p_1$ ) lies outside $PA_{KC}(C)$ , so it is replaced by the closest end point of $PA_{KC}(C)$ , this is ( $p'_1$ ). $p_2$ and $p_3$ points are the projections of opening centroids $oc_2$ and $oc_3$ on $PA_{KC}(C)$ , respectively. . . . .	45
3.8	Skeletonization using (a) only PA, (b) all principal axes. . . . .	46
3.9	If all opening centroids connect to one point which is close to one end of the $PA_{KC}$ then we connect this point with the $PA_{KC}$ end point on the other side of the component. . . . .	49
3.10	This example shows a rotation refinement on a component with one opening centroid. (Left) shows the initial principal axis orientation and (right) is illustrated the produced skeleton after alignment execution. . . . .	50
3.11	(a) The covariance-computed principal axes of the component. (b) The first refinement execution: The yellow axis is closer to the given vector, so it will be the new principal axis. (c) The second refinement execution: Again, the yellow axis is closer to the given vector. (d) The last refinement execution: The pink axis is closer to the given vector, so it will be the new principal axis (The initial and the new principal axes is now the same). (e) The difference of the extracted skeletons, before and after the refinement. . . . .	52

3.12	Skeleton extraction without (a) and with (b) using the Parent Refinement Algorithm. . . . .	52
3.13	Oriented bounding boxes and skeletons of “Human” components: (left) Original configuration (right) After alignment. . . . .	53
4.1	A framework for skeletal animation of articulated figures. . . . .	55
4.2	Ball and Socket joint. . . . .	57
4.3	A continuous curve is fit through the keys provided by the animator. . . .	58
4.4	Form and evaluation of (a) flat, (b) linear, and (c) smooth Hermite tangents rules. . . . .	59
4.5	(a) Constant value, (b) linear, and (c) cyclic channel extrapolation modes.	60
4.6	Keyframing animation file format. . . . .	61
4.7	Parent axis vector. . . . .	63
4.8	Forward kinematics example. . . . .	64
4.9	Bounding sphere collision example. . . . .	64
4.10	The forward kinematics skinning framework. . . . .	65
4.11	This is an example which shows the difference of testing with and without opening plane (red lines). If we are testing the parent points with the child’s OBB without the opening plane, then it will result to remove the green and orange points. On the other hand, the cut is more accurate since the removed points are only the green ones. . . . .	67
4.12	This is an example which shows the point removing process. We start from the opening points (yellow) and continue with their adjacents until we find an outside point. When we have found all end points (orange) then we maintain their neighbor information (purple) for use in the triangulation method. . . . .	68
4.13	This is an example which shows the point classification. Group 1 is the Circle end point set, Group 2 is the Bezier end point set, Group 3 is the Extra point set, and finally the Group 4 is the Removed point set. . . . .	69
4.14	Introducing additional points and blending curves. . . . .	70
4.15	(Left) A rejected facet. (Right) The normal evaluation of all possible point cases. Component facets are shown in green color, deleted facets are shown in grey and constructed facets are shown in pink color. . . . .	71
5.1	The main graphical user interface: We see a horse model rendered by its vertices that facilitate the view of its skeleton representation. . . . .	74
5.2	Cow model skeleton representation using (left) Opening and (right) Centroid methods. . . . .	78
5.3	Cow model skeleton representation using only principal axis (left) without and (right) with Local Refinement (12 degrees) . . . . .	78
5.4	Cow model skeleton representation using all principal directions (left) without and (right) with Parent Refinement . . . . .	79

5.5	Homer model skeleton representation using (left) Opening and (right) Centroid methods. . . . .	80
5.6	Homer model skeleton representation using only principal axis (left) without and (right) with Local Refinement (18 degrees) . . . . .	80
5.7	Homer model skeleton representation using all principal directions (left) without and (right) with Parent Refinement . . . . .	81
5.8	Dilo model skeleton representation using (up) Opening and (down) Centroid methods. . . . .	82
5.9	Dilo model skeleton representation using only principal axis (up) without and (down) with Local Refinement (10 degrees) . . . . .	82
5.10	Dilo model skeleton representation using all principal directions (left) without and (right) with Parent Refinement . . . . .	82
5.11	Camel model skeleton representation using (left) Opening and (right) Centroid methods. . . . .	83
5.12	Camel model skeleton representation using only principal axis (left) without and (right) with Local Refinement (22 degrees) . . . . .	83
5.13	Camel model skeleton representation using all principal directions (left) without and (right) with Parent Refinement . . . . .	84
5.14	Horse model skeleton representation using (up) Opening and (down) Centroid methods. . . . .	85
5.15	Horse model skeleton representation using only principal axis (up) without and (down) with Local Refinement (20 degrees) . . . . .	85
5.16	Horse model skeleton representation using all principal directions (left) without and (right) with Parent Refinement . . . . .	86
5.17	Dino model skeleton representation using (left) Opening and (right) Centroid methods. . . . .	87
5.18	Dino model skeleton representation using only principal axis (left) without and (right) with Local Refinement (18 degrees) . . . . .	87
5.19	Dino model skeleton representation using all principal directions (left) without and (right) with Parent Refinement . . . . .	88
5.20	A sequence of images obtained from applying a variety of motions on Cow model. . . . .	92
5.21	A sequence of images obtained from applying a variety of motions on Homer model. . . . .	93
5.22	First sequence of images obtained from applying a variety of motions on Dilo model. . . . .	94
5.23	Second sequence of images obtained from applying a variety of motions on Dilo model. . . . .	96
5.24	A sequence of images obtained from applying a variety of motions on Camel model. . . . .	97

5.25 A sequence of images describing the steps of the gap reconstruction process. The first picture shows the starting pose of a man's knee. The second one shows the knee's orientation after the use of FK skinning system. The third picture shows how the components are after removing the skin vertices. The fourth picture shows the computed patch vertices (Blue: *Circle*(down) and *Bezier*(up) group, Red: *End* points, Green: *Extra* group ) which reconstruct the gap. Final result (after the triangulation and computation of normals processes completed) is illustrated at the final picture. . . . . 98

# LIST OF TABLES

---

2.1	Comparison of memory usage. . . . .	31
2.2	Comparison of memory usage. . . . .	31
2.3	Comparison of operation counts for decomposition. . . . .	32
2.4	Comparison of operation counts for transforming one vector. . . . .	32
2.5	Comparison of operation counts for transforming n vectors. . . . .	33
5.1	Experimental models. . . . .	76
5.2	Time performance of skeletonization methods. Columns from left to right represent time performance using Opening method, Centroid method, Principal Axis method using only the covariance-computed PA and our Principal Axis method. . . . .	77
5.3	Time performance of pre-processing functions. Columns from left to right represent time performance computing median length, components which did not belong to checking node's sub-tree and neighbors for all vertices. . . . .	90
5.4	Time performance of Gap construction system animating 4 components of Cow model. . . . .	90
5.5	Time performance of Gap construction system animating 4 components of Homer model. . . . .	90
5.6	Time performance of Gap construction system animating 4 components of Dilo model. . . . .	91
5.7	Time performance of Gap construction system animating 4 components of Camel model. . . . .	91



# LIST OF ALGORITHMS

---

1	<code>principal_axis(<math>C</math>)</code> . . . . .	46
2	<code>group(<math>OC, PA_{KC}</math>)</code> . . . . .	48
3	<code>local_refine(<math>C, OC, KC, PD, Root</math>)</code> . . . . .	51
4	<code>parent_refine(<math>PD^C, PD^P</math>)</code> . . . . .	51
5	<code>advanced_rigid_skinning()</code> . . . . .	72

# ABSTRACT

---

Vasilakis, Andreas-Alexandros, T., S. MSc,

Computer Science Department, University of Ioannina, Greece. July, 2008.

Thesis Title: Robust Skeletal Animation of Articulated Modular Solid Objects

Thesis Supervisor: Fudos Ioannis

In video games, crowd simulations, computer generated imagery films and other applications of 3D computer graphics, skin motion of deformable objects in a realistic-looking way is of utmost importance. Skeleton-based skinning is widely used for plausible animation of complex characters defining mesh movement as a function of the underlying skeleton. A number of approaches have been employed in recent years based on efficient variations of the linear blend skinning (LBS) method. Such techniques usually require an appropriate weight selection process which is often based on a training set of example poses. In this dissertation, we propose a novel robust skeleton-based animation framework of articulated modular solid objects.

The contribution of this work is twofold. First, we present refinement techniques for improving the skeletal representation based on local characteristics which are extracted using centroids and principal axes of the components of the character. Skeleton-based animation is then performed using forward kinematics and quaternions. The components position varies over time, guided by an animation controller. Then, we use rigid skinning deformations by assigning each skin vertex one driver bone to achieve realistic skin motion avoiding vertex weights. A novel method eliminates the artifacts caused by self-intersections, especially in areas near joints, providing sufficiently smooth skin deformation. Finally, we have implemented all the above steps and we perform an extensive experimental evaluation of our suite of techniques with respect to efficiency, robustness and quality of the final animation outcome.

# ΕΚΤΕΤΑΜΕΝΗ ΠΕΡΙΛΗΨΗ ΣΤΑ ΕΛΛΗΝΙΚΑ

---

Ανδρέας-Αλέξανδρος Βασιλάκης του Θεοφίλου και της Σοφίας.

Msc, Τμήμα Πληροφορικής Πανεπιστημίου Ιωαννίνων, Ελλάδα. Ιούλιος 2008.

Εύρωστες Τεχνικές για την Κίνηση Αρθρωτών Αντικειμένων Βασισμένες στη Εξαγωγή Προσεγγιστικών Σκελετών.

Επιβλέπωντας: Ιωάννης Φούντος.

Στην βιομηχανία του κινηματογράφου και των βιντεοπαιχνιδιών όπως και σε εφαρμογές 3D γραφικών, η απόδοση κίνησης στερεών αρθρωτών αντικειμένων είναι ύψιστης σημασίας. Η τεχνική κινηματικής με χρήση σκελετού χρησιμοποιείται ευρέως για την δημιουργία αληθοφανών κινήσεων ορίζοντας την κίνηση της επιφανείας ως συνάρτηση της κίνησης του σκελετού που βρίσκεται στο εσωτερικό της. Τα τελευταία χρόνια πολλές παραλλαγές της κλασικής μεθόδου *linear blend skinning* (LBS) έχουν προταθεί. Όμως, αυτές οι τεχνικές συνήθως απαιτούν μια κατάλληλη διεργασία υπόλογισμού βαρών που κυρίως βασίζεται σε ένα σύνολο εκπαίδευσης από διάφορες πόζες του μοντέλου. Σ' αυτήν την εργασία, προτείνουμε μια νέα τεχνική άκαμπτης κινηματικής στερεών αρθρωτών μοντέλων με χρήση σκελετού.

Αρχικά, παρουσιάζουμε τεχνικές βελτίωσης με βάση τοπικά χαρακτηριστικά της αναπαράστασης του σκελετού που έχει εξαχθεί χρησιμοποιώντας κέντρα και κύριους άξονες των τμημάτων του μοντέλου. Το υποσύστημα εμπρόσθιας κινηματικής (FK) είναι υπεύθυνο για την κίνηση των τμημάτων του μοντέλου χρησιμοποιώντας *quaternions*. Τα τμήματα αλλάζουν θέση υπό τον έλεγχο ενός τμήματος λογισμικού που ονομάζεται ελεγκτής κίνησης. Κατόπιν, προτείνουμε μία μέθοδο επίλυσης του προβλήματος της ομαλής συνένωσης των επιφανειών των τμημάτων στις περιοχές κοντά στις αρθρώσεις. Η μέθοδος χωρίζεται στα παρακάτω βήματα:

1. Αφαίρεση των κορυφών επιφανείας γειτονικών κινούμενων τμημάτων.
2. Εύρεση νέων σημείων στην περιοχή για να συμπληρώσουμε τα κενά που δημιουργήθηκαν.
3. Κατασκευή “μπαλώματος” (patch), τριγωνοποιώντας τα νέα σημεία.
4. Υπολογισμός των κάθετων διανυσμάτων στις κορυφές επιφανείας.

Τέλος, περιγράφουμε τα αποτελέσματα της υλοποίησης και πειραματικής επαλήθευσης της ευρωστίας και αποδοτικότητας των μεθόδων που αναπτύξαμε.

# CHAPTER 1

## INTRODUCTION

---

1.1 Definition of the Problem

1.2 Applications

1.3 Related Work

1.4 Structure of Thesis

---

*Animation* is derived from the Latin *anima* and means the act, process, or result of imparting life, interest, spirit, motion, or activity. Motion is a defining property of life and much of the true art of animation is about how to tell a story, show emotion, or even express subtle details of human character through motion. “Animating” is moving something which can’t move itself. Animation adds to graphics the dimension of time which vastly increases the amount of information which can be transmitted. In order to animate something, the animator has to be able to specify, either directly or indirectly, how the object is to move through time and space. A key issue is to create animation tools which are expressive enough for the designer to specify what he wants while at the same time are powerful enough to provide for the automatic reconstruction of motion details. The appropriateness of a particular animation tool depends on the effect desired by the animator. An artistic animation clip will require different tools than an animation intended to simulate reality.

A computer is a secondary tool for achieving these goals, it is a tool which a skilful animator can use to help get the result he wants faster and without concentrating on technicalities in which he is not interested. Animation without computers, which is now often called “traditional” animation, has a long and rich history of its own which is continuously being written by hundreds of people still active in this art. As in any established field, some time-tested rules have been crystallized which give general high-level guidance to how certain things should be avoided. These principles of “traditional” animation apply equally to computer animation.

The computer, however, is more than just a tool. In addition to making the animator's main task less tedious, computers also add some truly unique abilities that were simply not available or were extremely difficult to obtain before. Modern modeling tools allow the relatively easy creation of detailed three-dimensional models, rendering algorithms can produce an impressive range of appearances, from fully photorealistic to highly stylized, powerful numerical simulation algorithms can help to produce desired physics-based motion for particularly hard to animate objects and motion capture systems give the ability to record and use real-life motion. These developments led to an exploding use of computer animation techniques in motion pictures and commercials, automotive design and architecture, medicine and scientific research among other areas. Completely new domains and applications have also appeared including fully computer-animated feature films 1.1, virtual/augmented reality systems and, of course, computer games [96].



Figure 1.1: Toy Story is a 1995 Pixar Animation Studios [107] film using Computer Animation.

There are two main categories of computer animation: *computer-assisted animation* and *computer generated animation*. Computer-assisted animation usually refers to 2D and 2 1/2 dimensional systems that computerize the traditional animation process. Interpolation between key shapes is typically the only algorithmic use of the computer in the production of this type of animation (in addition to the more “non-animation” uses of the computer such as inking, implementing a virtual camera stand, suffling paper, and managing data).

On the other hand, motion specification for computer-generated animation is divided into two categories: Low level techniques (techniques that aid the animator in precisely specifying motion), and High level techniques (techniques used to describe general motion behaviour).

Low level techniques consist of techniques, such as shape interpolation algorithms

(in-betweening), which help the animator fill in the details of the motion once enough information about the motion has been specified by the animator. When using low level techniques, the animator usually has a fairly specific idea of the exact motion that he or she wants.

High level techniques are typically algorithms or models used to generate a motion using a set of rules or constraints. The animator sets up the rules of the model, or chooses an appropriate algorithm, and selects initial values or boundary values. The system is then set into motion, so to speak, and the motion of the objects is controlled by the algorithm or model. The model-based/algorithmic approaches often rely on fairly sophisticated computation, such as physically based motion control.

In practice, of course, these categories are the extremes which characterize a continuum of approaches. Any technique requires a certain amount of effort from the animator and a certain amount of effort from the computer. One of the things which distinguish animation techniques is whether it's the animator or the computer which bears most of the burden. Motion specification aids are those techniques which seem to require more input from the user and fairly straightforward computation. Model-based approaches, on the other hand, require less from the animator and more computation. But the categories are really artificial. Approaches which I call motion specification aids could be discussed as algorithmic approaches which happen to require more input from the user.

Another way to characterize the difference between techniques is to look at the level of abstraction at which the animator is working. In one extreme, at a very low level of abstraction, the animator could color in every pixel individually in every frame. At the other extreme, at a very high level of abstraction, the animator could tell a computer to "make a movie about a dog". Presumably, the computer would whirl away while it computes such a thing. A high level of abstraction frees the animator from dealing with all of the details. A low level of abstraction allows the animator to be very precise in specifying exactly what is displayed when. In reality, animators want to be able to switch back and forth and work at various levels of abstraction. The challenge to developing animation tools is designing the tools so that animators are allowed to work at high levels of abstraction when desired, while providing them the ability to work at low levels when needed.

## 1.1 Definition of the Problem

This thesis studies the problem of the skeletal animation of solid articulated figures. *Skeletal animation*, sometimes referred to as *rigging*, is a technique in computer animation, particularly in the animation of vertebrates, in which the object's model consists of at least two main layers. The motion of a highly detailed surface representing the outer shell or skin of the object is what the viewer will eventually see in the final product. The *skeleton* underneath it is a hierarchical structure of joints which provides a kinematic

model of the figure and is used for animation. Skeletal animation technique can be divided into sub-problems which consist of the following distinct levels:

- Skeletonization

As the name implies, we build a skeleton inside the meshes we wish to animate by constructing hierarchy bones which is associated with some portion of the object's visual representation.

- Keyframing animation

Instead of animating the mesh itself we animate the skeleton within. Skeletal animation produced from key frames which are built by the animator in modeling software or digitally recording action movements of moving objects (*motion capture*). A sequence of key frames defines which movement the spectator will see, whereas the position of the key frames on the animation defines the timing of the movement. There are two techniques in computer graphics to animate an articulated object:

1. Forward Kinematics

The essential concept of this technique is that the positions of particular parts of the model at a specified time (framework) are calculated from the position and orientation of the object, together with any information on the joints of the articulated model.

2. Inverse Kinematics

This technique refers to a process that calculates the required articulation of a series of joints, such that the end effectors of specified bones end up in a particular location. In contrast to forward kinematic animation, where each movement for each component must be planned, only the starting and ending locations of the bone are necessary.

- Skin Deformation

A skeleton acts as a special type of deformer transferring its motion to the skin by assigning each skin vertex one (*rigid skinning*) or more (*linear blending skinning*) joints as drivers. In the former case, a skin vertex is fixed in the local coordinate space of the corresponding bone following whatever motion this bone is subjected to. Although simple, this technique suffers from inherent flaws from self-intersections, especially in areas around joints. In the latter case of linear blending skinning (LBS) (also known as *skeleton subspace deformation (SSD)*, *vertex blending* or *enveloping*), each skin vertex is assigned multiple influences and blending weights for each joint. This scheme provides more detailed control over the results. The deformed vertex is

computed by a convex combination of individual vertex transformations. Although the method is simple and easy to implement in GPUs, the generated meshes exhibit volume loss as joints are rotated to extreme angles producing non-natural deformations (also known as “collapsing joint” and “candy wrapper” effects, see Figure 1.2). Despite these drawbacks, variations of this method are widely used in interactive computer graphics applications.

In this paper, we present an integrated skeleton-based rigid skinning framework for animating 3D solid modular articulated models. A visually satisfactory skeleton is extracted using centroids and principal axes [63] of the components of a segmented object by performing a depth-first traversal of the skeleton hierarchy tree. Furthermore, we refine the produced skeleton segments with local and neighbor features to derive better skeletal representations that are more appropriate for our application. Skeleton based animation is then performed using the forward kinematics and quaternions. The joints position varies over time, guided by an animation controller. Then, we use classic rigid skinning by assigning each skin vertex to an influence bone to achieve mesh animation avoiding thus vertex weight estimation and training pose set production costs. We introduce a novel method to eliminate degeneracies and artifacts from self-intersections, especially in areas near joints, by performing alternative intuitive deformations. Briefly, our approach first removes the skin vertices of the overlapping component parts and then adds new vertices to fill in the gap. Finally, for each frame it constructs a blending mesh that produces a smooth surface using a robust triangulation method.

## 1.2 Applications

Skeletal animation is the standard way to animate characters or mechanical objects for a prolonged period of time (usually over 100 frames). It is commonly used by video game artists and in the movie industry, and can also be applied to mechanical objects and any other object made up of rigid elements and joints. Motion capture can speed up development time of skeletal animation, as well as increasing the level of realism. For motion that is too dangerous for motion capture, there are computer simulations that automatically calculate physics of motion and resistance with skeletal frames. Virtual anatomy properties such as weight of limbs, muscle reaction, bone strength and joint constraints may be added for realistic bouncing, buckling, fracture and tumbling effects known as virtual stunts. Virtual stunts are controversial due to its potential to replace stunt performers. However, there are other applications of virtual anatomy simulations such as military and emergency response. Virtual soldiers, rescue workers, patients, passengers and pedestrians can be used for training, virtual engineering and virtual testing of equipment. Virtual anatomy technology may be combined with artificial intelligence for further enhancement of animation and simulation technology.



## 1.3 Related Work

There is an abundance of research work in the literature that tackles the skeleton extraction and skinning of 3D objects from different perspectives. Our work lies at the intersection of two large bodies of literature: Skeletonization and skinning. In this review, we focus on recent developments most closely related to animation.

### 1.3.1 Skeletonization

Skeletonization algorithms may be classified based on whether they work on the boundary surface (*geometric methods*) or on the inner volume (*volumetric methods*).

**Geometric methods.** *Medial Axis Transform (MAT)* [18] is a popular topological skeletal representation technique which consists of a set of curves which roughly run along the object’s middle. MAT-based representations suffer from perturbation and noise dependence, high computation cost for 3D [26] ( $O(n^2 \log n)$  in worst case [76]), shape complexity (because in 3D they contain not only lines but also surface elements). All these characteristics makes them inappropriate for animation applications. Researchers have proposed approximate MAT using Voronoi diagram [4, 29] or dual Delaunay triangulation [8]. *Reeb graphs* are a fundamental 1-D data structure for representing the configuration of critical points and their relationships in an attempt to capture the intrinsic topological shape of the object [77, 82, 73, 12]. However, a remeshing technique [9, 44] is usually required to generate accurate skeletons.

**Volumetric methods.** Several methods generate skeletons by constructing discrete field functions by means of the object’s volume. Many functions have been proposed to solve the problem such as distance transform [84], repulsive force field [64, 25] and radial basis [89]. Other volumetric-based techniques make use of a multi-resolution thinning process applied on the model’s voxelized representation [37, 65]. Although accurate, such methods are usually very time consuming and they cannot be applied in animation since the volumetric information needs to be computed since usually it is not being given as part of the model to be animated.

**Mesh-based methods.** [75] presents a method for extracting a hierarchical, rigid skeleton from a set of example poses. Researchers also generate skeletons based on mesh contraction [60, 11]. Moreover, [50] extract a skeleton using a hierarchical mesh decomposition algorithm. [63] proposed an iterative approach that simultaneously generates hierarchical shape decomposition and a corresponding set of multi-resolution skeletons. Such method offer adequate accuracy and are quite efficient. We have refined a recent method [63] that was initially invented for reverse engineering for the purposes of character animation.

### 1.3.2 Skinning

Linear blend skinning (LBS) is the most widely used technique for real-time animation in spite of its limitations [87] (see Figure 1.2) due to its computational efficiency and straightforward GPU implementation [74, 62]. LBS was initially presented in a game development magazine [56, 57]. LBS determines the new vertex position by linearly combining the results of the vertex transformed rigidly with each influence bone. Most recent skinning algorithms are classified based on whether they use one (*Geometric methods*) or a training set of poses (*Example based methods*) of input models. In the first case, vertex weighting is usually specified manually. In the latter case, researchers have proposed automatically approximate authoring skins techniques by training weights with multiple input meshes [70, 49, 69].

**Geometric methods** revert to non-linear blending of rigid transformations since deformation inherently spherical. Numerous proposed methods have replaced the linear blending domain with simple quaternion [43, 2] matrix operator [68], log-matrix [24] operator, spherical blending [51] and dual quaternion [52]. Another alternative recently explore is the use of 3D free-form character articulation using sweep-based [47, 91] or spline-based techniques [90, 34]. This is a promising research direction but not yet versatile applicable. Our rigid skinning approach falls under this broad class of algorithms introducing a novel versatile, robust and efficient blending approach based on rational quadratic Bezier patches.

**Example-based methods** remove artifacts by correcting LBS errors with storage and computation cost increase. Initial approaches combined rigid skinning with interpolation examples using radial basis functions [59, 80]. EigenSkin [53] used principal component analysis for approximating the original deformation model. Multi-Weight Enveloping (MWE) [85] and Animation Space [69] are similar methods that add more weights per influence bone to provide more precise approximations and additional flexibility. [48] found the Animation Space technique to consistently perform better from LBS and MWE while MWE suffered from overfitting. In addition, [70] introduced extra bones to capture richer deformations than the standard LBS model. Finally, recent research work proposed a replacement of linear with rotational regression when examples are available [86, 88].

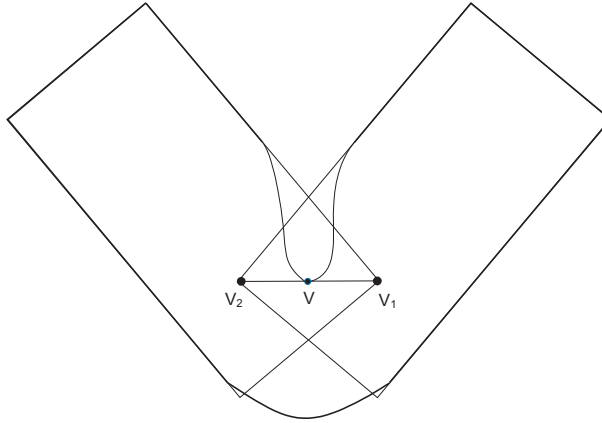


Figure 1.2: Illustration of skeleton subspace deformation algorithm. The deformed position of a point  $V$  lies on the line  $V_1V_2$  defined by the images of that point rigidly transformed by the neighboring skeletal coordinate frames, resulting in the characteristic “joint collapsing” problem.

## 1.4 Structure of Thesis

The rest of this thesis is organized as follows. Chapter 2 discusses fundamentals of computer graphics and computational geometry algorithms and tools that we use throughout our work. In Chapter 3 we focus on the skeletonization process. We study proposed methods and present modifications and refinements for more advanced and efficient skeleton extraction. Chapter 4 proposes an integrated framework for robust skeletal animation of articulated modular solid objects. Chapter 5 discusses the development of the three-dimensional skeletal animation framework and discusses the experiment setup that was used to drive quantitative and qualitative results of the proposed algorithms. Finally, Chapter 6 concludes with a summary of our contribution and identification of future research directions.

# CHAPTER 2

## BACKGROUND MATERIAL

---

2.1 Vector and Tools

2.2 Transformations

2.3 Computational Geometry Techniques

2.4 Data Structures for geometric objects

---

In computer graphics, we work with objects defined in 3D. All objects to be drawn, as well as the cameras used to render them, have shape, position, and orientation. The two fundamental mathematical disciplines that come to our aid in graphics are *vector tools* and *transformations*. Furthermore, the research in combinatorial Computational geometry develop efficient algorithms and data structures for solving problems stated in terms of basic geometrical objects. So, by studying these disciplines in detail, we develop methods to describe the various geometric objects we will encounter, and we learn how to convert geometric ideas to numbers.

### 2.1 Vector and Tools

#### 2.1.1 Vector

Vector arithmetic provides a unified way to express geometric ideas algebraically. In graphics, we work with vectors of two, three, and four dimensions, but most definitions in this chapter apply to higher dimensions as well.

From a geometrical viewpoint, vectors are objects having length and direction. They represent various physical entities, such as velocity, force, and displacement. A vector is

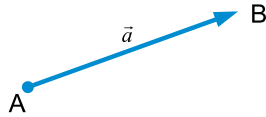


Figure 2.1: A vector going from A to B.

often drawn as an arrow of a certain length pointing in a certain direction. It is valuable to think a vector geometrically as a displacement from one point to another.

Points and vectors are defined with respect to some coordinate system. Figure 2.2 shows the coordinate systems that are normally used. Each system has an *origin* called  $O$  and some axes emanating from  $O$ . These axes are usually oriented at right angles to one another. Coordinates are marked along each axis and a point is assigned coordinates according to how far along it lays from each axis. Part(a) shows the usual two-dimensional system and Part(b) shows a *left-handed* and a *right-handed* 3D coordinate system.

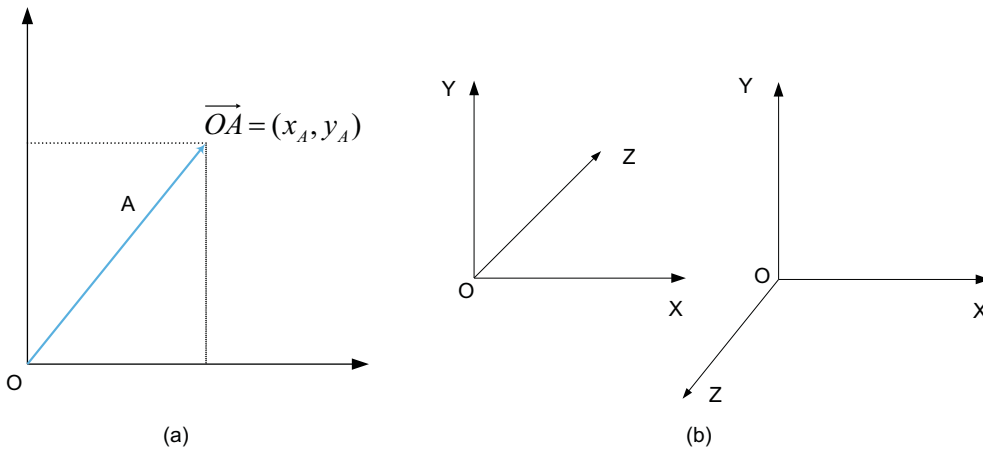


Figure 2.2: Coordinate systems. Part(a) The two-dimensional coordinate system. Part(b) The three-dimensional coordinate systems: The left-handed orientation is depicted on the left, and the right-handed is depicted on the right.

## Vector Operations

We may define for vectors most of the usual arithmetic operations that we associate with real numbers. Two vectors are equal if and only if they have the same length and direction. Also we define two fundamental operations for vectors: addition and multiplication by a scalar. Two vectors are added according to the parallelogram rule. This rule states that the sum of two vectors is found by placing the tail of either vector against the head of the other. The sum vector is the vector that completes the triangle started by the two vectors. The parallelogram is formed by taking the sum in either order (Figure 2.3).

We can scale a vector by multiplying it by a real number  $k$ . This just multiplies the vector's length by  $k$  without changing its direction. Also, in the case of  $k = -1$ , we create

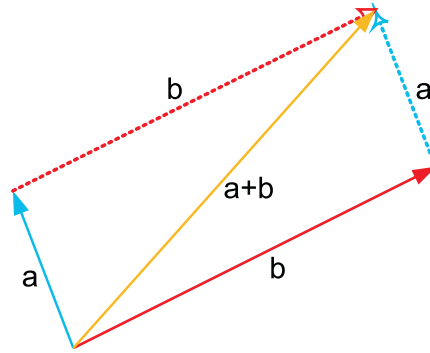


Figure 2.3: Vector addition: Parallelogram rule.

a unitary minus for a vector:  $-a$ . This is just a vector with the same length and opposite direction (Figure 2.4).

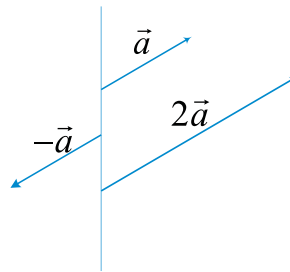


Figure 2.4: Vector scaling.

Subtraction follows easily once adding and scaling have been established:  $a - c$  is simple  $a + (-c)$ . Figure 2.5 shows the geometric interpretation of this operation, forming the difference of  $a$  and  $b$  as the sum of  $a$  and  $-b$ . With the parallelogram rule, this sum is seen to be equal to the vector that emanates from the head of  $c$  and terminates at the head of  $a$ .

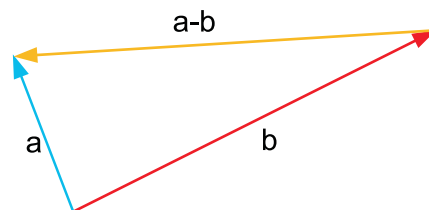


Figure 2.5: Vector subtracting.

## Linear Combinations of Vectors

With methods in hand for adding and scaling vectors, we can define a linear combination of vectors. To form a linear combination of two vectors  $v$  and  $w$  (having the same direction),

we scale each of them by some scalars, say,  $a$  and  $b$ , and add the weighted versions to form the new vector,  $av + bw$ . The more general definition for combining  $n$  such vectors are as follows: A linear combination of the  $n$  vectors  $v_1, v_2, \dots, v_n$  is a vector of the form  $w = a_1v_1 + a_2v_2 + \dots + a_nv_n$ , where  $a_1, a_2, \dots, a_n$  are scalars.

### The magnitude of a Vector - Unit Vectors

If a vector  $v$  is represented by the n-tuple  $(v_1, v_2, \dots, v_n)$ , we denote the magnitude by  $|v|$  and define it as the distance from the tail to the head of the vector. Using euclidian distance, we obtain  $|v| = \sqrt{v_1^2 + v_2^2 + \dots + v_n^2}$ .

It is often useful to scale a vector so that the result has unity length. This type of scaling is called *normalizing* a vector, and the result is known as unit vector. For example, we form the normalized vector of  $a$ , denoted by  $\hat{a}$ , by scaling  $a$  with the value  $\frac{1}{|a|} : \hat{a} = \frac{1}{|a|}a$ . Clearly,  $|\hat{a}| = 1$ , and  $\hat{a}$  is a unit vector having the same direction as  $a$  (Figure 2.6).

Finally, the *zero vector* is the vector of zero length. The direction of the zero vector is undefined.

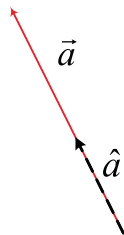


Figure 2.6: Normalizing a vector.

### Product of Vectors

Two other powerful tools that facilitate working with vectors are the *dot product* and the *cross product*. The dot product produces a scalar; the cross product operates only on 3D vectors and produces a vector.

The simplest way to multiply two vectors is the dot product. The dot product of  $a$  and  $b$  is denoted  $a \cdot b$  and is often called *scalar product* because it returns a scalar. The definition of the dot product generalizes easily to  $n$  dimensions:  $D = v \cdot w = \sum v_i w_i$ . The dot product returns a value related to its arguments length and the angle  $\phi$  between them (Figure 2.7):  $a \cdot b = |a||b| \cos \phi$ . The most important application of the dot product is the finding the angle between two vectors or between two intersecting lines:  $\phi = \arccos \frac{a \cdot b}{|a||b|}$ .

The dot product can also be used to find the projection of one vector onto another (Figure 2.8). This is the length  $a \rightarrow b$  of a vector  $a$  that is projected at right angles onto a vector  $b$ :  $a \rightarrow b = |a| \cos \phi = \frac{a \cdot b}{|b|}$ .

The sign of the dot product is used in many algorithmic tests. The case in which the vectors are  $90^\circ$  apart, or perpendicular, is of special importance. Recall that  $\cos \phi$  is

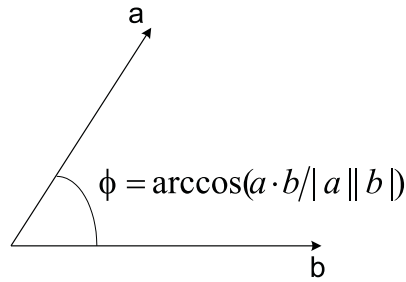


Figure 2.7: The dot product of two vectors.

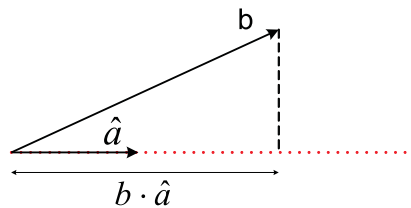


Figure 2.8: Scalar projection of a vector  $b$  in the direction of a vector  $a$ .

positive if  $|\phi|$  is less than  $90^\circ$ , zero if  $|\phi|$  equals  $90^\circ$ , and negative if  $|\phi|$  exceeds  $90^\circ$ . Because the dot product is proportional to the cosine of the angle between them, we can observe immediately that two vectors are

- Less than  $90^\circ$  if  $a \cdot b > 0$
- Exactly  $90^\circ$  if  $a \cdot b = 0$
- More than  $90^\circ$  if  $a \cdot b < 0$

The cross product of two three dimensional vectors returns a 3D vector that is perpendicular to the two arguments of the cross product (Figure 2.9). Given the 3D vectors  $a = (a_x, a_y, a_z)$  and  $b = (b_x, b_y, b_z)$ , their cross product is denoted as  $a \times b$ . It is defined in terms of the standard unit vectors  $i, j,$  and  $k$  by  $a \times b = (a_y b_z - a_z b_y)i + (a_z b_x - a_x b_z)j + (a_x b_y - a_y b_x)k$ . So, in coordinate form  $a \times b = (a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x)$

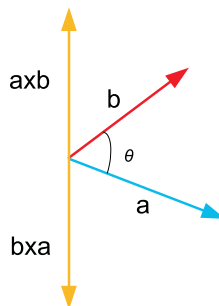


Figure 2.9: The cross product of two vectors.



## 2.1.2 2D Implicit Curves

Intuitively, a curve is a set of points that can be drawn on a piece of paper without lifting the pen. A common way to describe a curve is using an *implicit equation*. An implicit equation in two dimensions has the form  $f(x, y) = 0$ . The function  $f(x, y)$  returns a value. Points where this value is zero are on the curve, and points where the value is non-zero are not on the curve. Because we can multiply an implicit equation by any constant  $k$  without changing the points where it is zero,  $kf(x, y) = 0$  is the same for any non-zero  $k$ .

### Line

The usual "slope-intercept" representation of the line is  $y = mx + b$ , which can be easily converted to implicit form:  $y - mx + b = 0$ . However this representation does not capture lines parallel to the  $y$  axis such as  $x = k$  because the "slope"  $m$  would then go to infinity. For this reason, a more general form is often useful:  $Ax + By + Cz = 0$ .

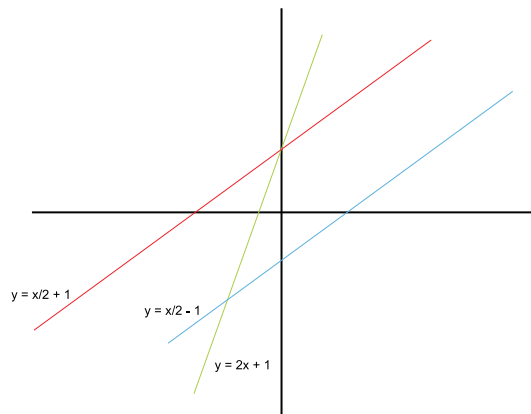


Figure 2.10: Three implicit lines.

## 2.1.3 2D Parametric Curves

A parametric curve is controlled by a single parameter that can be considered a sort of index that moves continuously along the curve. Such curves have the form  $(x, y) = (g(t), h(t))$ . Here  $(x, y)$  is a point in the curve, and  $t$  is the parameter that influences the curve. For a given  $t$ , there will be some point determined by the functions  $g$  and  $h$ . For continuous  $g$  and  $h$ , a small change in  $t$  will yield a small change in  $x$  and  $y$ . Thus, as  $t$  continuously changes, points are swept out in a continuous curve. This is a nice feature because we can use the parameter  $t$  to explicitly construct points on the curve.

## Line

A parametric line in 2D that passes through points  $p_0 = (x_0, y_0)$  and  $p_1 = (x_1, y_1)$  can be written as  $p = [x, y] = [x_0 + t(x_1 - x_0), y_0 + t(y_1 - y_0)]$ . Because the formulas for  $x$  and  $y$  have the same structure, we can use the vector form for  $p = (x, y) : p(t) = p_0 + t(p_1 - p_0)$ .

A nice feature on this form is that  $p(0) = p_0$  and  $p(1) = p_1$ . Since the point changes linearly with  $t$ , the value of  $t$  between  $p_0$  and  $p_1$  measures the fractional distance between the points.

Another parametric formaly for describing lines is:  $p(t) = o + td$ , where  $o$  is the starting point and  $d$  the direction vector. If the vector  $d$  has unit length, the line is arc-length parameterized. This means  $t$  is an exact measure of distance along the line.

One useful application is given two arc-length parameterized lines find the intersection point between them (Figure 2.11). Let's describe the technique and algorithm for determining the intersection point of two lines in 2D. It is obvious that the intersection point lies on both lines. To determine the point we have to compute the exact distance of this point from the starting point of one line in the direction of the line.

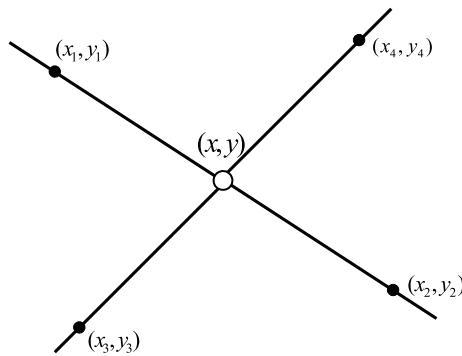


Figure 2.11: Line-Line intersection.

If our equations of the two lines are:  $p_1(t_1) = o_1 + t_1d_1$  and  $p_2(t_2) = o_2 + t_2d_2$  then to find the intersection point we have to solve the equation  $p_1(t_1) = p_2(t_2)$ . Solving gives the following two equations with two unknowns ( $t_1$  and  $t_2$ ):

$$\begin{aligned} o_1x + t_1d_1x &= o_2x + t_2d_2x \\ o_1y + t_1d_1y &= o_2y + t_2d_2y \end{aligned}$$

Moreover, solving gives the following expressions for  $t_1$  and  $t_2$ :

$$\begin{aligned} t_2 &= \frac{(d_1x(o_1y - o_2y) + d_1y(o_2x - o_1x))}{(d_1xd_2y - d_1yd_2x)} \\ t_1 &= \frac{(o_2x - o_1x + t_2d_2x)}{d_1x} \end{aligned}$$

Finally, by estimating the  $t_1$  and  $t_2$  we find the exact measure distance of intersection point along the two lines. That means that the intersection point parametric form is:

$$InterP = o_2 + d_2 \frac{((d_1x(o_1y - o_2y) + d_1y(o_2x - o_1x))}{(d_1xd_2y - d_1yd_2x)}.$$

## Hermite

Polynomials are functions of the canonical form:  $f(t) = a_0 + a_1t_1 + \dots + a_nt_n = \sum a_it_i$ . The  $a_i$  are called the coefficients and is called the degree of the polynomial. We generalize this form to  $f(t) = \sum c_ib_i(t)$  where  $b_i(t)$  is a polynomial and we call them *basis functions*. A very useful form of a cubic polynomial is the *Hermite* form. Hermite curves are very easy to calculate but also very powerful. They are used to smoothly interpolate between key-points (like object movement in keyframing animation or camera control). Hermite curves are convenient because they provide local control over the shape, and provide  $C^1$  continuity. However, since the user must specify both positions and derivatives, a special interface must be provided.

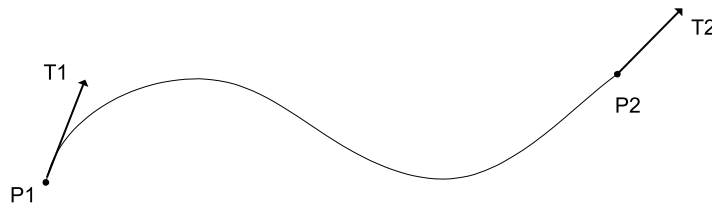


Figure 2.12: Hermite Curve.

In order to calculate a point on the hermite you have to find first the coefficients of the curve. To calculate the coefficients of the hermite curve you need the following vectors: the start point of the curve  $P_1$ , the tangent to how the curve leaves the start point  $T_1$ , the endpoint of the curve  $P_2$ , and the tangent to how the curve meets the endpoint  $T_2$  (Figure 2.12). In order to make it easier to understand can be expressed all this stuff with some vector and matrix algebra. Thus, to estimate the coefficients we construct a vector  $C$  that contains these 4 parameters vertices and multiply this by the hermite's basis matrix  $H$ :

$$C = [P_1, T_1, P_2, T_2]^T,$$

$$H = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$Coe f = [a, b, c, d]^T = H \cdot C$$

Finally, the equation for the constructions of points in the hermite curve have the form:  $P = T \cdot H \cdot C = T \cdot Coef$ , where the vector  $T$  is the interpolation-point and it's powers up to 3 ( $T = [t^3, t^2, t^1, 1]$ ). In order to increase this equation's performance speed, we can write it at the following form:  $P = d + t(c + t(b + t(a)))$ . Remember, this assumes that the parameter  $t$  varies from 0 to 1.

## Rational Bezier

*Bezier* curves are widely used in computer graphics to model free-form smooth curves. A Bezier curve is a polynomial curve that approximates its control points (Figure 2.13). The curves can be a polynomial of any degree. A quadratic Bezier curve is the path traced by the function  $B(t)$ , given the control points  $P_0$ ,  $P_1$ , and  $P_2$ :

$$B(t) = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2 \Rightarrow$$

$$B(t) = \sum_{i=0}^n P_i b_{i,n}(t)$$

where the polynomials  $b_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}$ ,  $i = 0, \dots, n$  are known as *Bernstein polynomials* of degree  $n$ .

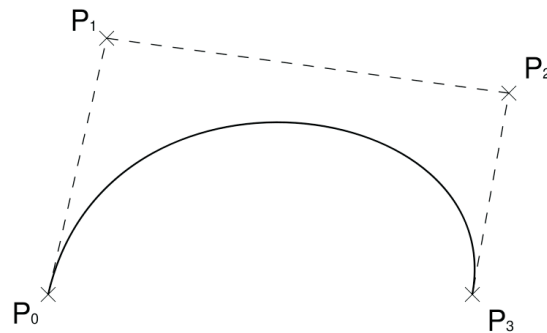


Figure 2.13: Cubic Bezier Curve.

The *Rational Bezier* adds adjustable weights to provide closer approximations to arbitrary shapes. The conic sections can be represented exactly by a ratio of two quadratic polynomials. The numerator is a weighted Bernstein-form Bezier curve and the denominator is a weighted sum of Bernstein polynomials. Given  $n$  control points  $P_i$ , the rational Bezier curve can be described by:

$$B(t) = \frac{\sum_{i=0}^n b_{i,n}(t) P_i w_i}{\sum_{i=0}^n b_{i,n}(t) w_i}$$

Fudos and Hoffman at [35] described how to construct blending arcs from constraints, using a unified rational parametric representation that combines the separate cases of blending parallel and non-parallel edges. Let  $v, u$  be the two tangents of the conic arc at the endpoints. Let  $C$  and  $D$  be the two endpoints of the conic arc, and let  $P$  be the third point we wish to interpolate. Furthermore, let  $U = (v_x M2 - u_x M1, v_y M2 - u_y M1)$ , where  $M1 = C_x v_y - C_y v_x$  and  $M2 = D_y u_y - D_x u_x$ . They prove that the rational parametric solution has the form:

$$c(t) = \frac{(1-t)^2 C + 2t(1-t)W + t^2 D}{(1-t)^2 + 2wt(1-t) + t^2}.$$

where

$$W = \frac{Area(C, P, D)U}{\sqrt{d(D, \vec{v}, C)d(P, \vec{v}, C)d(C, \vec{u}, D)d(P, \vec{u}, D)}}$$

$$w = \frac{Area(C, P, D)(\vec{v} \times \vec{u})}{\sqrt{d(D, \vec{v}, C)d(P, \vec{v}, C)d(C, \vec{u}, D)d(P, \vec{u}, D)}}$$

$$d(S, \vec{r}, R) = (S_y - R_y)r_x - (S_x - R_x)r_y$$

$Area(C, P, D)$  : The signed area of the triangle  $\widehat{CPD}$ .

and the formula is valid whatever the vector  $u$  and  $v$  are linearly dependent or not. An acceptable solution exists if, and only if,

- $d(D, \vec{v}, C)d(P, \vec{v}, C) > 0$
- $d(C, \vec{u}, D)d(P, \vec{u}, D) > 0$
- $w > -1$

### 2.1.4 3D Implicit Surfaces

Implicit equations implicitly define a set of points that are on the same surface  $f(x, y, z) = 0$ . Any point  $(x, y, z)$  that is on the surface returns zero when given as an argument to  $f$ . Any point not on the surface returns some number other than zero. This is called implicit rather than explicit because you can check whatever a point is on the surface.

#### Plane

A plane is a two-dimensional doubly ruled surface spanned by two linearly independent vectors. The generalization of the plane to higher dimensions is called a hyper-plane. The equation of a plane with non-zero normal vector  $n = (a, b, c)$  through the point  $p_0 = (x_0, y_0, z_0)$  is  $n \cdot (p - p_0) = 0$ , where  $p = (x, y, z)$ . Plugging in gives the general equation of a plane,  $ax + by + cz + d = 0$ , where  $d = -ax_0 - by_0 - cz_0$  (Figure 2.14).

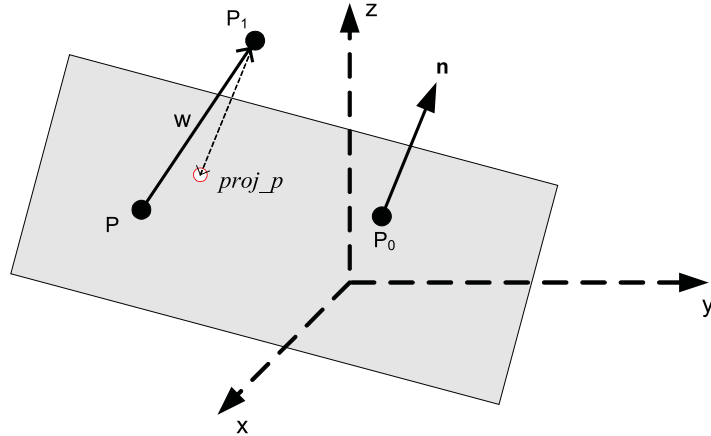


Figure 2.14: Point-plane distance and projection of point  $P_1$  on the plane.

Given a plane  $ax + by + cz + d = 0$  and a point  $p_1 = (x_1, y_1, z_1)$ , then the normal to plane is given by  $n = (a, b, c)$  and a vector from the plane to the point is given by  $w = p_1 - p$ , where  $p = (x, y, z)$ . Projecting  $w$  onto  $n$  we find that the signed distance  $D$  from the point to the plane as  $D = n \cdot p_1 + d$  which is positive if  $p_1$  is on the same side of the plane as the normal vector  $n$  and negative if it is on the opposite side. We can define, moreover, the *projection point* of  $p_1$  on the plane as:  $Proj_p = p_1 - n \cdot D$  (Figure 2.14)

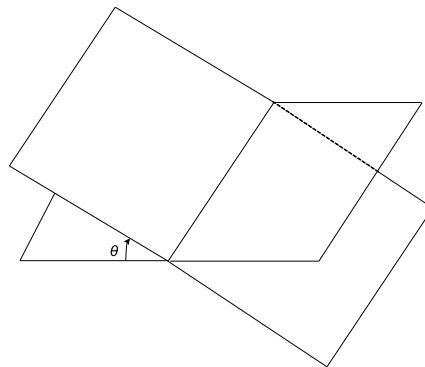


Figure 2.15: The dihedral angle between two planes.

The angle between two intersecting planes is known as the *dihedral angle* (Figure 2.15). The dihedral angle between the planes which have normal vectors  $n_1 = (a_1, b_1, c_1)$  and  $n_2 = (a_2, b_2, c_2)$  is simply given via the dot product of the normals,  $\cos \theta = n_1 \cdot n_2$ .

### 2.1.5 3D Parametric Surfaces

Another way to specify surfaces in 3D space is with two-dimensional parameters. These surfaces have the form:  $x = f(u, v), y = g(u, v), z = h(u, v)$ .

## Circle

A circle is one of the basic shapes of Euclidian geometry. It is the locus of all points in a plane at a constant distance, called the *radius*, from a fixed point, called the *center*. Through any three points not on the same line, there passes one and only one circle (Figure 2.16).

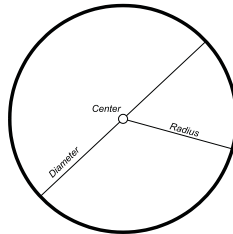


Figure 2.16: Circle.

If we know the center  $C$  and the radius  $R$  of a circle, and three points  $P_1$ ,  $P_2$ , and  $P_3$  on the circle we can find that the parametric form of the circle equation in three-dimensional space is:  $P = C + R \cos \theta U + R \sin \theta V$ , where  $N$  is the unit normal vector of the plane,  $U$  is the unit vector from the center toward a point on the circle and  $V$  is the cross product of  $N$  and  $U$ . Or we can write it by  $x, y, z$  component:

$$\begin{aligned}P_x &= C_x + R \cos \theta U_x + R \sin \theta V_x \\P_y &= C_y + R \cos \theta U_y + R \sin \theta V_y \\P_z &= C_z + R \cos \theta U_z + R \sin \theta V_z\end{aligned}$$

where

$$\begin{aligned}C &= P_1 + \frac{W}{2|(P_2 - P_1) \times (P_3 - P_1)|^2} \\R &= \frac{|W|}{2|(P_2 - P_1) \times (P_3 - P_1)|^2} \\W &= |P_3 - P_1|^2[(P_2 - P_1) \times (P_3 - P_1)] \times (P_2 - P_1) + \\&\quad |P_2 - P_1|^2[(P_3 - P_1) \times (P_2 - P_1)] \times (P_3 - P_1)\end{aligned}$$

## Triangle

Triangles in both 2D and 3D are the fundamental modeling primitive in many graphics programs. A triangle is one of the basic shapes of geometry: a polygon with three vertices and three sides or edges which are line segments. In Euclidean geometry any three non-collinear points determine a unique triangle and a unique plane.

The normal vector to a triangle can be found by taking the cross product of any two vectors in the plane of the triangle. It is easiest to use two of the three edges as the vectors, for example,  $n = (c - b) \times (c - a)$  (Figure 2.17). Note that this normal vector is not necessarily of unit length.

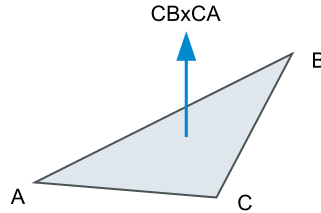


Figure 2.17: Normal of a triangle.

The not signed area of the triangle can be found by taking the length of the cross product:  $Area = \frac{1}{2}|(b - a) \times (c - a)|$ . This area will have positive sign and the normal obeys the right-hand rule if the points a, b, c are in counter-clockwise order, and a negative sign, otherwise.

Often we wish to assign a property at each triangle vertex and smoothly interpolate the value of that property across the triangle. The simplest way to do this is the *barycentric coordinates*. First let us consider a triangle  $T$  defined by three vertices  $A_1$ ,  $A_2$  and  $A_3$ . Any point  $P$  located on this triangle may then be written as a weighted sum of these three vertices:  $P = t_1A_1 + t_2A_2 + t_3A_3$ , where  $t_1$ ,  $t_2$  and  $t_3$  are the barycentric coordinates (Figure 2.18). These are subjected to the constraint  $t_1 + t_2 + t_3 = 1$ . If we weight all vertices with the same weight  $t_1 = t_2 = t_3 = 1/3$  then the resulting point is located in the center of the triangle:  $(A_1 + A_2 + A_3)/3$ .

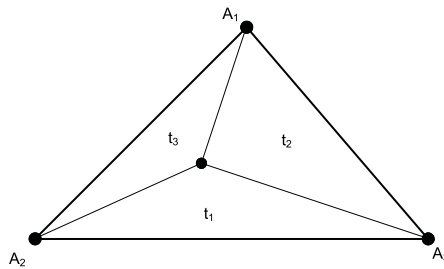


Figure 2.18: Barycentric coordinates.

## 2.2 Transformations

A transformation, or mapping, in elementary terms is any of a variety of different functions from geometry, such as rotations, translations, and scalings. These can be carried out in



Euclidean space, particularly in dimensions 2 and 3. They are also operations that can be performed using linear algebra, and explicitly using matrix and quaternion theory.

An *affine transformation* or *affine map* between two vector spaces (strictly speaking, two affine spaces) consists of a linear transformation followed by a translation:  $x \mapsto Ax + b$ . In the finite-dimensional case each affine transformation is given by a matrix  $A$  and a vector  $b$ , which can be written as the matrix  $A$  with an extra column  $b$ . In general, an affine transform is composed of zero or more linear transformations (rotation, scaling or shear) and translation (shift). Several linear transformations can be combined into a single matrix, thus the general formula given above is still applicable.

Ordinary vector algebra uses matrix multiplication to represent linear transformations, and vector addition to represent translations. Using a trick, it is possible to represent both using matrix multiplication. The trick requires that all vectors are augmented with a "1" at the end, and all matrices are augmented with an extra row of zeros at the bottom, an extra column - the translation vector - to the right, and a "1" in the lower right corner. If  $A$  is a matrix,

$$\begin{bmatrix} \vec{y} \\ 1 \end{bmatrix} = \begin{bmatrix} A & \vec{b} \\ 0, \dots, 0 & 1 \end{bmatrix} \begin{bmatrix} \vec{x} \\ 1 \end{bmatrix}$$

is equivalent to the following  $\vec{y} = A\vec{x} + \vec{b}$ . Ordinary matrix-vector multiplication always maps the origin to the origin. Since the set of vectors with 1 in the last entry does not contain the origin, translations within this subset using linear transformations are possible. This is the *Homogenous coordinates* system and the advantage of using it, is that one can combine any number of affine transformations (rotations, translations, scales, etc) into one by multiplying the matrices.

### 2.2.1 3D Matrix Theory

In this section, we describe how we can use matrix multiplications to accomplish changes in a vector such as scaling, rotation, and translation. We will also discuss how these transforms operate differently on points (displacement vectors: points are represented as offset vectors from the origin), and normal vectors.

We use coordinate frames and suppose that we have an origin  $O$  and three mutually perpendicular axes in the direction  $i$ ,  $j$ , and  $k$ . Point  $P$  in this frame is given by  $P = O + P_x i + P_y j + P_z k$  and so has the representation  $P = (P_x, P_y, P_z, 1)$ . Now suppose  $T$  is an affine transformation that transforms point  $P$  to point  $Q$ . Then,  $T$  is represented by a 4 by 4 matrix:

$$M = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and we can say that the representation of point  $Q$  is found by multiplying  $P$  by  $M$ :  $Q = M \cdot P$ . We must notice that for an affine transformation, the final row of the matrix is a string of zeroes followed by a lone 1.

## Translation

If we want to move a vector from its current location to somewhere else, in technical jargon, we call this a translation; we must use matrix form

$$M = \begin{bmatrix} 0 & 0 & 0 & m_{14} \\ 0 & 0 & 0 & m_{24} \\ 0 & 0 & 0 & m_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Check that  $Q = MP$  is simply a shift in  $Q$  by the vector  $m = (m_{14}, m_{24}, m_{34})$ .

## Rotation

There is a much greater variety of rotations in three than in two dimensions, since we must specify an axis about which the rotation occurs, rather than just a single point.

The simplest rotation is a rotation around one of the coordinate's axes. We call a rotation around x-axis an *x-roll*, a rotation around the y-axis a *y-roll* and a rotation around z-axis a *z-roll*. We present individually the matrices that produce a x-roll, a y-roll and a z-roll. In each case, the rotation is through an angle  $\theta$ , around the given axis. ( $c$  stands for  $\cos(\theta)$  and  $s$  for  $\sin(\theta)$ )

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, R_y(\theta) = \begin{bmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, R_z(\theta) = \begin{bmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

## Composing Transformations

The process of applying several transformations in succession to form one overall transformation is called *composing* the transformations. Not surprisingly, 3D affine transformations can be composed, and the result is another 3D affine transformation. The matrix that represents the overall transformation is the product of the individual matrices  $M_1$  and  $M_2$  that perform the two transformations, with  $M_2$  premultiplying  $M_1$ :  $M = M_2 M_1$ . Notice that the transformations appear in reverse order to that in which the transformations are applied. By applying the same reasoning, any number of affine transformations can be computed simply by multiplying their associated matrices. In this way, transformations based on an arbitrary succession of rotations, scaling, shears and translations can be formed and captured in a single matrix.

## Euler Angles

According to Euler's rotation theorem, any rotation (or sequence of rotations) around a point is equivalent to a single rotation around some axis through that point. This asserts that any rotation may be obtained by three rolls about the x-, y-, and z-axes using three angles  $(\theta_1, \theta_2, \theta_3)$ . The three angles giving the three rotation matrices are called *Euler angles*. There are several conventions for Euler angles (there are 12 possible ways), depending on the axes about which the rotations are carried out.

Because of its historical popularity, computer animation systems were quick to use Euler angles as parameters for animating orientation. There are two major draw-backs to this approach, however. The first is a practical problem often encountered by animators trying to set up an arbitrary orientation using Euler angles, and the second is a mathematically deep objection to their use when interpolating orientation (*Gimbal Lock*). Both of these problems occur because Euler angles ignore the interaction of the rolls about the separate axes (Figure 2.19). This is and one major reason why we use the quaternion representation.

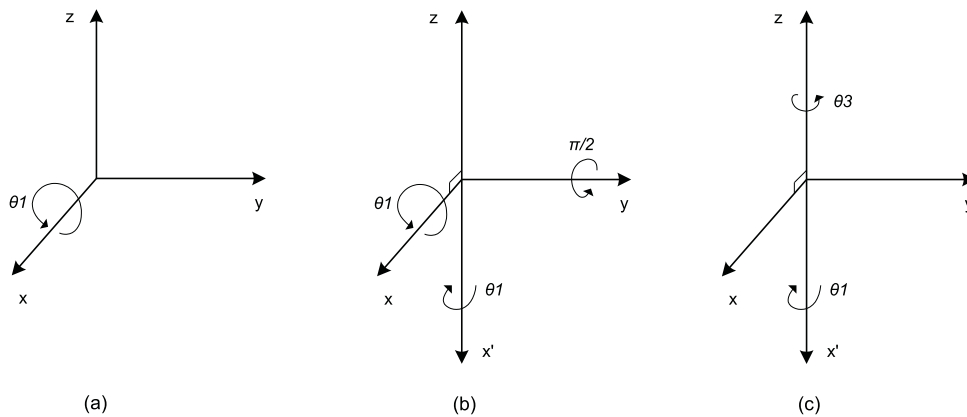


Figure 2.19: Illustrating the loss of one degree of freedom (Gimbal Lock). Part(a) x-roll  $\theta_1$ . Part(b) x-roll  $\theta_1$  followed by y-roll  $\pi/2$ . x-axis effectively gets rotated to  $x'$ -axis. Part(c) followed by z-roll  $\theta_3$ . z-roll  $\theta_3$  same as x-roll  $-\theta_1$ .

## Examples

- *Rotating about an arbitrary point.*

So far, all of the rotations we have considered have been about the origin. But suppose we wish instead to rotate points about some other point in the space. To achieve this operation, we must relate the rotation about the desired point to an elementary rotation about origin.

If the desired point is  $V = (v_x, v_y, v_z)$  and the rotation axis and angle are  $U$  and  $\theta$ , then first of all, we translate all points so that  $V$  coincides with the origin, then a rotation

about the origin will be appropriate. Once this rotation is done, then the whole scene is shifted back to restore  $V$  to its original location. The rotation therefore consists of the following three elementary transformations:

1. Translate point  $P$  through vector  $-V = (-v_x, -v_y, -v_z)$ .
2. Rotate around  $U$  vector about the origin through angle  $\theta$ .
3. Translate  $P$  back through vector  $V$ .

So, creating a matrix for each elementary transformation and multiplying the matrices out, products the transformation matrix:  $M = T(V)R_u(\theta)T(-V)$

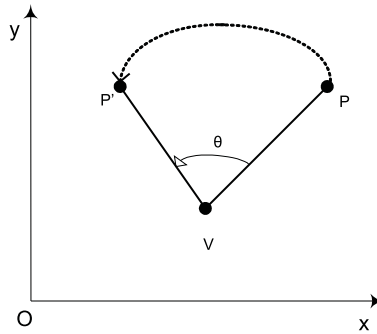


Figure 2.20: Rotating vector  $P$  about an arbitrary point  $V$

- *Alignment of vector with a unit vector  $k$  which lies on the positive z-axis.*

With the help of this example we can coincide any given matrix on space with one of the basic axes  $X, Y, Z$ . So we can exalt the problem of rotating around an arbitrary axis to the problem of rotating around the x- or y- or z-axis.

In order to find the transformation matrix that aligns one vector at a unit vector that lies on positive z-axis, we must estimate the two angles  $\theta_1$  and  $\theta_2$  according to the Figure 2.21. In other words, the following steps are required to find the requested matrix:

1. Rotation of the vector  $v$  around x-axis through angle  $\theta_1$  in order the vector  $v$  to fall on the positive side of XZ plane (Figure 2.21b).
2. Rotation of the vector  $v_1$  around y-axis through angle  $\theta_2$  in order the vector  $v_1$  to fall on the positive side of z-axis(Figure 2.21c).

But we have not computed the angles  $\theta_1$  and  $\theta_2$ . From Figure 2.21 we observe that  $\theta_1$  can be estimated by the angle which created from  $v$  projection on  $\widehat{YZ}$  plane with z-axis (we assume that the  $b$  and  $c$  are both not zero). From the triangle  $\widehat{OP'B}$  we have:  $\sin \theta_1 = \frac{b}{\sqrt{b^2+c^2}}$  and  $\cos \theta_1 = \frac{c}{\sqrt{b^2+c^2}}$ . Furthermore, for the calculation of the angle  $\theta_2$  from the triangle  $\widehat{OQ'Q}$  we have:  $\sin \theta_2 = \frac{a}{\sqrt{a^2+b^2+c^2}}$  and  $\cos \theta_2 = \frac{\sqrt{b^2+c^2}}{\sqrt{a^2+b^2+c^2}}$ .

Finally, after finding the angles we can create a matrix for each elementary transformation and by multiplying these matrices out products:  $M = R_y(\theta_2)R_x(\theta_1)$ , which leads at:

$$M = \begin{bmatrix} \frac{\lambda}{|\vec{v}|} & \frac{-ab}{|\lambda\vec{v}|} & \frac{-ac}{|\lambda\vec{v}|} & 0 \\ 0 & \frac{c}{\lambda} & \frac{-b}{\lambda} & 0 \\ \frac{a}{|\vec{v}|} & \frac{b}{|\vec{v}|} & \frac{c}{|\vec{v}|} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \lambda = \sqrt{b^2 + c^2}.$$

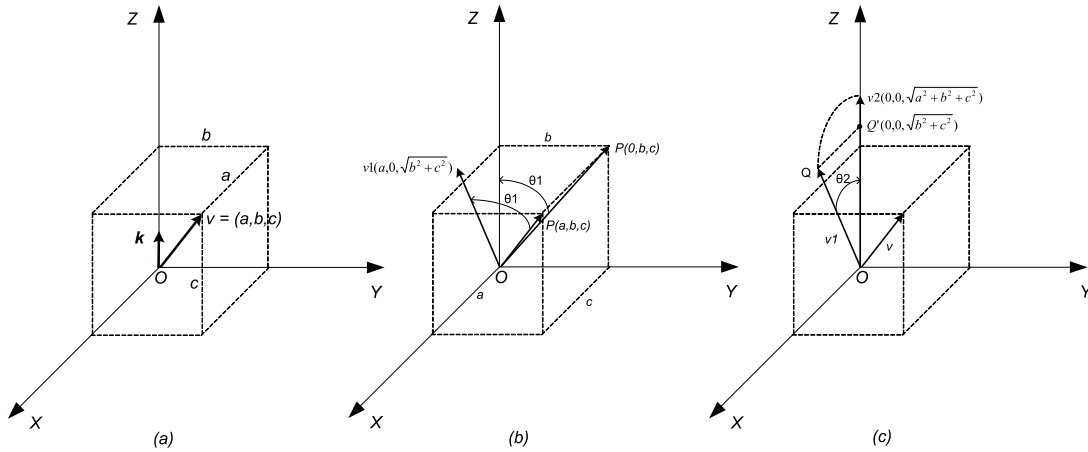


Figure 2.21: Alignment of vector with a unit vector  $k$  which lies on positive z-axis

## Inverse of Matrix

While we can always invert a matrix algebraically, we can use geometry if we know what the transformation does. For example, the inverse of scale  $(s_x, s_y, s_z)$  is scale  $(1/s_x, 1/s_y, 1/s_z)$ . The inverse of a rotation is its transpose. The inverse of a translation is a translation in the opposite direction. In addition, if we have a series of matrices  $M = M_1 M_2 \dots M_n$  then  $M^{-1} = M_n^{-1} \dots M_2^{-1} M_1^{-1}$ .

Interestingly, we can write the transformation matrix  $M$  without scaling in the form  $M = RT$ , where  $R$  and  $T$  are  $4 \times 4$  matrices. As we described above, the inverse of a rotation  $R$  is its transpose:  $R^T$  and the inverse of a translation  $T$  is a translation in the opposite direction:  $-T$ . So, from the rules above it follows easily that:  $M^{-1} = R^T(-T)$ .

## Transforming Normal Vectors

While most of the 3D vectors we use, represents positions or directions, some vectors represent surface normals. Surface normal vectors are perpendicular to the tangent plane of a surface. These normals do not transform the way we would like when the underlying surface is transformed. We can derive a transform matrix  $N$  which does take

the normal to a vector perpendicular to the transformed surface. Because the normal is a direction vector, we don't want it to get the translation from the matrix, so we only need to multiply the normal by the upper  $3 \times 3$  portion of the rotation matrix  $M$ .

### 2.2.2 Quaternion Theory

*Quaternions* are a non-commutative extension of complex numbers [98]. Although they have been superseded in most applications by vectors and matrices, they still find uses in both theoretical and applied mathematics; in particular, they provide a convenient mathematical notation for representing orientations and rotations of objects in three dimensions.

Compared to Euler angles they are simpler to compose and avoid the problem of Gimbal Lock. Moreover, compared to rotation matrices they are more efficient and more numerically stable and instead of specifying the nine elements of a matrix we define four real numbers. We begin by studying the angular displacement of a vector rotating a vector by  $\theta$  about an axis  $n$ .

#### Angular Displacement

We define rotation as an angular displacement given by  $(\theta, n)$  of an amount  $\theta$  about an axis determined by a normalized vector  $n$ . Consider the angular displacement on a vector  $r$  taking it to position  $Rr$  as shown in Figure 2.22.

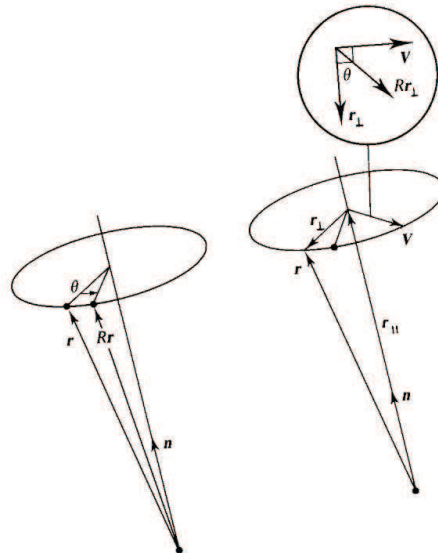


Figure 2.22: Angular displacement  $(\theta, n)$  of  $r$

The problem can be decomposed by resolving  $r$  into components parallel to  $n$ , which by definition remains unchanged after rotation, and perpendicular to  $n$  in the plane passing through  $r$  and  $Rr$ :  $r_{\parallel} = (n \cdot r)n$  and  $r_{\perp} = r - (n \cdot r)n$ , and  $r_{\perp}$  is rotated into position

$Rr_{\perp}$ . We construct a vector perpendicular to  $r_{\perp}$  and lying in the plane. To compute this rotation, we use:  $V = n \times r_{\perp} = n \times r$ , so  $Rr_{\perp} = r_{\perp} \cos \theta + V \sin \theta$ , hence

$$\begin{aligned} Rr &= r_{\parallel} + Rr_{\perp} \\ &= r_{\parallel} + r_{\perp} \cos \theta + V \sin \theta \\ &= (n \cdot r)n + (r - (n \cdot r)n) \cos \theta + (n \times r) * \sin \theta \\ &= (\cos \theta)r + (1 - \cos \theta)n(n \cdot r) + (\sin \theta)n \times r. \end{aligned}$$

## Quaternions and spatial rotation

We will show that rotating the vector  $r$  by the angular displacement can be achieved by a quaternion transformation. We began by noting that the effect such an operation we will only need four real numbers. We require:

- the change of length of the vector
- the plane of the rotation (which can be defined by two angles from two axes)
- the angle of the rotation

In other words, we need a representation that only possesses the four degrees of freedom required to Euler's theorem. For this we will use *unit quaternions*. As the name implies, quaternions are "four vectors" and can be considered as a generalization of complex numbers with  $s$  as the real or scalar part and  $x, y, z$  the imaginary part:

$$q = s + xi + yj + zk = (s, (x, y, z)) = (s, v).$$

A quaternion can specify a point or vector in four-dimensional space. If  $s = 0$ , a point or vector in the three-dimensional space. In this context they used to represent a vector plus rotation.  $i, j, k$  are unit quaternions and are equivalent to unit vectors in a vector system; however, they obey different combination rules:  $i^2 = j^2 = k^2 = -1$  and  $ij = k$ ,  $ji = -k$ .

## Quaternion Operations

Using the above rules we can derive the following rules, each of which yields a quaternion:

1. Addition:  $q_1 + q_2 = (s_1 + s_2, v_1 + v_2)$
2. Multiplication:  $q_1q_2 = (s_1s_2 - v_1 \cdot v_2, v_1 \times v_2 + s_1v_2 + s_2v_1)$
3. Conjugate:  $q' = (s, -v)$

4. Magnitude:  $qq' = s^2 + |v|^2 = q^2$ , if equals to 1 then  $q$  is called a *unit quaternion*.

5. Rotation:

The set of all unit quaternions forms a unit sphere in four-dimensional space and unit quaternions play an important part in specifying general rotations. It can be shown that if we have a quaternion  $q = (s, v)$  then exists a  $v'$  and a  $\theta \in [-\pi, \pi]$  such that  $q = (\cos \theta, v' \sin \theta)$  and if  $q$  a unit quaternion then  $q = (\cos \theta, n \sin \theta)$  where  $|n| = 1$ . We now consider operating on a vector  $r$  in Figure 2.22 by using quaternions.  $R$  is defined as the quaternion  $p = (0, r)$  and we define the operation as :  $R_p(p) = qpq^{-1}$ .

That is, it is proposed to rotate the vector  $r$  by expressing it as a quaternion, multiplying it on the left by  $q$  and on the right by  $q^{-1}$ . This guarantees that the result will be a quaternion of the form  $(0, v)$ , in other words a vector.  $q$  is defined to be a unit quaternion  $(s, v)$ . It is easily shown that:

$$\begin{aligned} R_p(p) &= (0, (s^2 - v \cdot v)r + 2v(v \cdot r) + 2s(v \times r)) \Rightarrow \\ Rq(p) &= (0, (\cos^2 \theta - \sin^2 \theta)r + 2 \sin \theta^2(v \cdot r) + 2 \cos \theta \sin \theta(v \times r)) \\ &= (0, r \cos 2\theta + (1 - \cos 2\theta)n(n \cdot r) + \sin 2\theta(n \times r)) \end{aligned}$$

Now comparing this with angular displacement equation above, we noticed that aside from a factor of 2 appearing in the angle they are identical in form. The act of rotating a vector  $r$  by angular displacement  $(\theta, n)$  is the same as taking this angular displacement, lifting it into quaternion space, by representing it as the unit quaternion  $(\cos \theta/2, n \sin \theta/2)$  and performing the operation  $q()q^{-1}$  on the quaternion  $(0, r)$ . We could therefore parameterize orientation in terms of the four parameters  $\cos \theta/2, n_x \sin \theta/2, n_y \sin \theta/2, n_z \sin \theta/2$  using quaternion algebra to manipulate the components.

We conclude rotation section by noting that quaternions are used exclusively to represent orientation; they can be used to represent translation but combining rotation and translation into scheme analogous to homogenous coordinates is not straightforward.

6. Moving out quaternion space:

The implementation of such a scheme requires us to move into and out of quaternion space, that is, to go from a general matrix to a quaternion and vice versa. It can be shown that if we want to rotate a vector with the quaternion  $q$ , is exactly equivalent to applying the following rotation matrix to the vector:



$$M = \begin{bmatrix} 1 - 2(y^2 + z^2) & 2xy - 2sz & 2sy + 2xz & 0 \\ 2xy + 2sz & 1 - 2(x^2 + z^2) & -sx + 2yz & 0 \\ -2sy + 2xz & sx + 2yz & 1 - 2(x^2 + y^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

By these means then, we can move from quaternion space to rotation matrices.

## 7. Decomposition into two same quaternions:

An extra operation we used in this work, is the decomposition of a quaternion  $q = (s, v)$  into the product of two same quaternions:  $q = q'q'$ , where  $q' = (s', v')$ . This is very useful when we have a rotation movement and we want to estimate a sequence of “keyframing” rotation movements that represent our rotation. This can be accomplished by executing this operation iteratively to the resulting quaternions. In order to evaluate this decomposition we write:

$$\begin{aligned} q &= (s, v) = q'q' = (s', v')(s', v') = \\ &= (s's' - v' \cdot v', v' \times v' + s'v' + s'v') = \\ &= ((s')^2 - ((v'x)^2 + (v'y)^2 + (v'z)^2), 2s'v') \Rightarrow \\ s' &= \sqrt{\frac{s + \sqrt{(s^2 - vx^2 - vy^2 - vz^2)}}{2}} \\ v' &= \frac{v}{2s'} \end{aligned}$$

## 8. Alignment of vector with a unit vector k which lies on positive z-axis:

Using the same solution process as we described above with matrix structure, we come to the two following steps which are required to find the requested quaternion:

- (a) Rotation of the vector  $v$  around x-axis through angle  $\theta_1$  in order the vector  $v$  to fall on the positive side of  $XZ$  plane:  $Q_x(\theta_1)$ .
- (b) Rotation of the vector  $v1$  around y-axis through angle  $\theta_2$  in order the vector  $v1$  to fall on the positive side of z-axis:  $Q_y(\theta_2)$ .

The two angles can be computed with the same equations discussed previously. So, after evaluating the angles  $\theta_1$  and  $\theta_2$ , we create a quaternion for each elementary rotation and multiplying these quaternions out products:  $Q = Q_x(\theta_1)Q_y(\theta_2)$ .

### 2.2.3 Performance Issues

At this section we compare which is the best representation to use for rotations between the rotation matrix  $R$  and the quaternion  $q$ . Various high level operations are compared by a count of low level operations including multiplication (M), addition or subtraction (A), division (D), and expensive math library function evaluations (F). Summary tables are provided to allow you to quickly compare the performance [104].

#### Memory Usage

In the memory usage comparison it is obviously clear that the rotation matrix uses more memory because it requires 9 float numbers than a quaternion which requires 4 floats.

Table 2.1: Comparison of memory usage.

representation	floats
$R$	9
$q$	4

#### Composition Time

The product of two rotation matrices requires 27 multiplications and 18 additions for a total cost of  $18A + 27M$ . The product of two quaternions requires 16 multiplications and 12 additions for a total cost of  $12A + 16M$ , clearly outperforming matrix multiplication. Moreover, renormalizing a quaternion to adjust for floating point errors is cheaper than renormalizing a rotation matrix using Gram-Schmidt orthonormalization.

Table 2.2: Comparison of memory usage.

representation	A	M
$R$	18	27
$q$	12	16

#### Decomposition Time

The decomposition of a rotation matrix requires 27 multiplications and 18 additions for a total cost of  $18A + 27M$ . The decomposition of a quaternion requires 4 multiplications, 5 additions, 4 divisions and 2 math library function (square functions) calls for a total cost of  $5A + 4M + 4D + 2F$ , clearly outperforming matrix splitting.

Table 2.3: Comparison of operation counts for decomposition.

representation	A	M	D	F
$R$	18	27		
$q$	5	4	4	2

## Transformation Time

The transformation of vector  $\vec{V}$  by a rotation matrix is the product  $\vec{U} = R\vec{V}$  and requires 9 multiplications and 6 additions for a total of 15 operations. If  $\vec{V} = (v_0, v_1, v_2)$  and if  $\vec{V} = v_0i + v_1j + v_2k$  is the corresponding quaternion with zero  $s$  component, then the rotate vector  $\vec{U} = (u_0, u_1, u_2)$  is computed as  $\vec{U} = u_0i + u_1j + u_2k = q\vec{V}q^*$ . Applying the general formula for quaternion multiplication directly, the product  $p = q\vec{V}$  requires 16 multiplications and 12 additions. The product  $p q^*$  also uses the same number of operations. The total operation count is 56. However, since  $\vec{V}$  has no  $s$  term,  $p$  only requires 12 multiplications and 8 additions—one term is theoretically zero, so no need to compute it. We also know that  $\vec{U}$  has no  $s$  term, so the product  $p q^*$  only requires 12 multiplications and 9 additions. Using these optimizations, the total operation count is 41. Observe that conversion from quaternion  $q$  to rotation matrix  $R$  requires 12 multiplications and 12 additions. Transforming  $\vec{V}$  by  $R$  takes 15 operations. Therefore, the process of converting to rotation and multiplying uses 39 operations, two less than calculating  $q\vec{V}q^*$ . Table 2.4 is a summary of the operation counts for transforming a single vector.

Therefore, the rotational formulation yields the fastest transforming. But keep in mind that a batch transform of  $n$  vectors requires converting the quaternion to a rotation matrix only once at a cost of 24 operations. The total operations for transforming by quaternion are  $24 + 15n$ . Table 2.5 is a summary of the operation counts for transforming  $n$  vectors.

Table 2.4: Comparison of operation counts for transforming one vector.

representation	A	M	comments
$R$	6	9	
$q$	24	32	using generic quaternion multiplies
$q$	17	24	using specialized quaternion multiplies
$q$	18	21	convert to matrix, then multiply

## 2.3 Computational Geometry Techniques

Computational geometry is a branch in computer science, which deals with the study of algorithms to solve problems stated in terms of geometry. The primary goal of research in combinatorial computational geometry is to develop efficient algorithms and

Table 2.5: Comparison of operation counts for transforming  $n$  vectors.

representation	A	M	comments
$R$	$6n$	$9n$	
$q$	$24n$	$32n$	using generic quaternion multiplies
$q$	$17n$	$24n$	using specialized quaternion multiplies
$q$	$12+6n$	$12+9n$	convert to matrix, then multiply

data structures for solving problems stated in terms of basic geometrical objects: points, line segments, polygons, polyhedra, etc. The kernel problems in computational geometry may be classified in different ways, according to various criteria. In the static problems, some input is given and the corresponding output needs to be constructed or found. We are interesting of two fundamental problems: Kernel's Centroid and Minimum Bounding Volume estimation.

### 2.3.1 Kernel's Centroid

The *kernel*  $K$  of a polyhedron  $P$  with  $n$  vertices is the locus of the points internal to  $P$  from which all vertices of  $P$  are visible, assuming that the polyhedron boundary is not transparent. Each facet of a polyhedron defines an interior *half-plane*, informally defined as a half-plane that contains interior points of the polygon in the vicinity of the edge in question. The kernel of a polyhedron is the intersection of all its interior half-planes (Figure 2.23). We use the *qhalf* algorithm, a routine of *qhull* library [108], which implements the intersection of the  $n$  interior half-planes in optimal  $\Theta(n \log n)$  time [95].

We may compute the *kernel's centroid* ( $CK$ ) using the average of all intersection points. However, this approach generally works unsatisfactory for kernels whose points are informally distributed over the model space. It is sufficient to note that centroid computation of a bounding volume is independent of clustering of object vertices. This can easily be seen by considering adding (or taking away) extra vertices off-center, inside or on the boundary, of a bounding volume. These actions do not affect the defining centroid of the volume and therefore should not affect its calculation. The situation can be improved by considering just extremal points, using only those points on the convex hull of the model. This eliminates the internal points, which can no longer mis-compute the centroid. So, the proposed algorithm is to compute the centroid of the convex hull as the mean of the triangle centroids weighted by their area [40].

Given  $n$  triangles  $(p_k, q_k, r_k)$ ,  $0 \leq k \leq n$ , in the convex hull, the centroid is given by  $m_H = \frac{\sum a_k m_k}{A_H}$ , where  $a_k = |(q_k - p_k) \times (r_k - p_k)|/2$  is the area,  $m_k = (p_k + q_k + r_k)/3$  is the centroid of the triangle  $k$ , and  $A_H = \sum a_k$  is the total area of the convex hull.

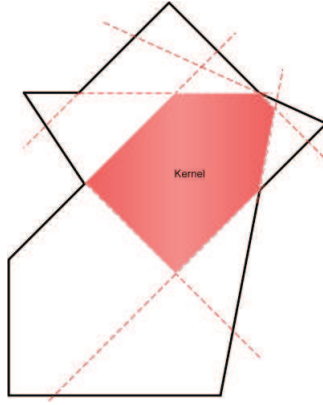


Figure 2.23: Kernel of a polygon is shown at this figure in red.

### 2.3.2 Minimum Bounding Volume

A *bounding volume* for a set of objects is a closed volume that completely contains the union of the objects in the set [93]. Bounding volumes are used to improve the efficiency of geometrical operations by using simple volumes to contain more complex objects. Normally, simpler volumes have simpler ways to test for overlap. Bounding volumes are most often used to accelerate certain kinds of tests.

In collision detection, when two bounding volumes do not intersect, then the objects cannot collide, either. Testing against a bounding volume is typically much faster than testing against the object itself, because of the bounding volume's simpler geometry. This is because an object is typically composed of polygons or data structures that are reduced to polygonal approximations.

The choice of the type of bounding volume for a given application is determined by a variety of factors: the computational cost of computing a bounding volume for an object, the cost of updating it in applications in which the objects can move or change shape or size, the cost of determining intersections, and the desired precision of the intersection test. It is common to use several types in conjunction, such as a cheap one for a quick but rough test in conjunction with a more precise but also more expensive type. We choose to use three fundamental volumes: Bounding Sphere, Oriented Bounding Box and Convex Hull.

#### Bounding Sphere

A *bounding sphere* is a sphere containing the object. In 2D graphics, this is a circle. Bounding spheres are represented by centre and radius. They are very quick to test for collision with each other: two spheres intersect when the distance between their centers does not exceed the sum of their radius. This makes bounding spheres appropriate for objects that can move in any number of dimensions (Figure 2.24).

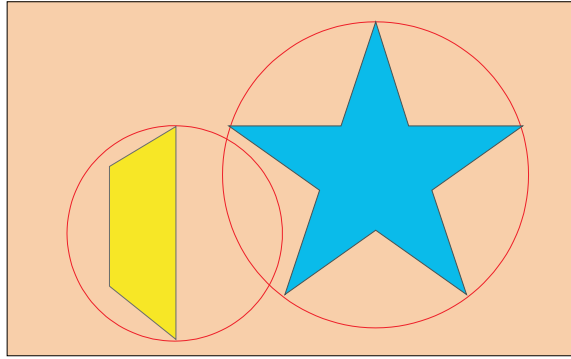


Figure 2.24: In a sphere-sphere intersection, the routine may report that collision has occurred when it really hasn't.

## Bounding Box

A *bounding box* is a cuboid, or in 2-D a rectangle, containing the object. In dynamical simulation, bounding boxes are preferred to other shapes of bounding volume such as bounding spheres for objects that are roughly cuboid in shape when the intersection test needs to be fairly accurate. In many applications the bounding box is aligned with the axes of the coordinate system, and it is then known as an *axis-aligned bounding box (AABB)*. Figure 2.25 shows AABBs and objects inside them. To distinguish the general case from an AABB, an arbitrary bounding box called *oriented bounding box (OBB)* is used.

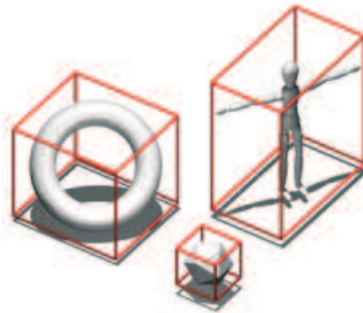


Figure 2.25: Objects and their axis-aligned box boundaries.

”Axis-aligned” refers to the fact that either the box is aligned with the world axes or each face of the box is perpendicular to one coordinate axis. This basic piece of information can cut down the number of operations needed to transform such a box. Again, the trade-off for speed is precision. Because AABBs always have to be axis-aligned, we can't just rotate them when the object rotates; they have to be recomputed for each frame. Still, this computation isn't difficult and doesn't slow us down much if we know the extents of each character model. However, we still face precision issues. For example, let's assume that we're spinning a thin, rigid rod in 3D, and we'd like to construct an AABB for each

frame of the animation. As we can see, the box approximates each frame differently and the precision varies (Figure 2.26).

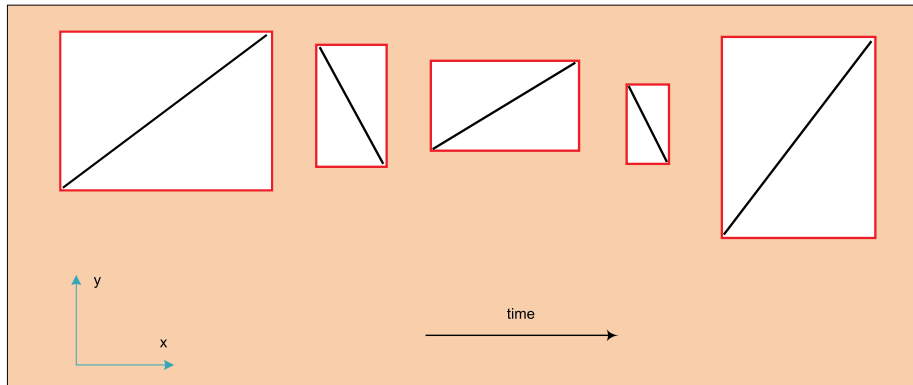


Figure 2.26: Successive AABBs for a spinning rod (as viewed from the side).

So, rather than use AABBs, we may use boxes that are arbitrarily oriented and minimize the empty space, or error, of the box approximation. This technique is based on what are called oriented bounding boxes (OBBs) and has been used for ray tracing and interference detection for quite some time. This technique is not only more accurate, but also more robust than the AABB technique. However, OBBs are lot more difficult to implement, slower, and inappropriate for dynamic or procedural models (an object that morphs, for instance). If we want to figure out fast if two oriented bounding boxes overlap, the *separating axis* theorem can be applied. This theorem tells us that one of the tests that we could perform would be to project the boxes on some axis in space and check whether the intervals overlap. If they don't, the given axis is called a separating axis (Figure 2.27).

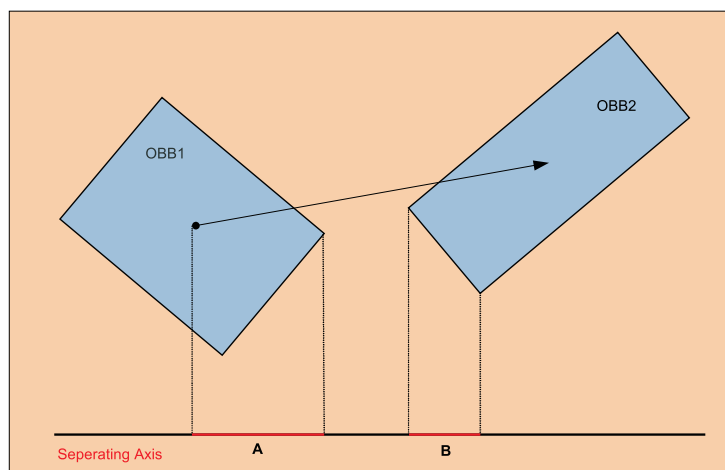


Figure 2.27: Separating axis (intervals do not overlap).

If overlap occurs on every single separating axis (there are only 15), the boxes intersect. Thus, it's very easy to determine whether or not two boxes intersect.

## Convex Hull

Finding the *convex hull* ( $CH$ ) of a set of points is the most elementary interesting problem in computational geometry. It arises because the hull quickly captures a rough idea of the shape or extent of a data set. This forms a unique polygon, where its vertices are the outside members of the set. The convex hull or convex envelope for a set of points  $X$  in a real vector space  $V$  is the minimal convex set containing  $X$ .

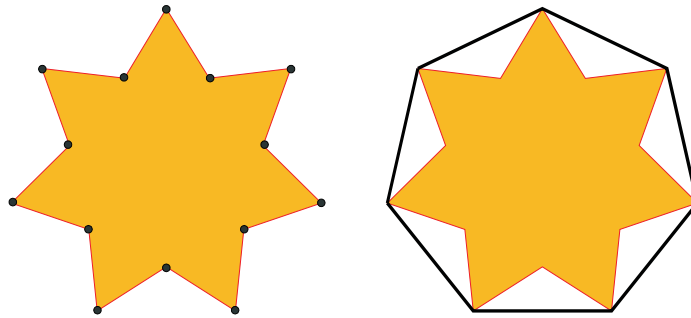


Figure 2.28: Convex hull of a set of 2D points.

In computational geometry, numerous algorithms are proposed for computing the convex hull of a finite set of points, with various computational complexities. Computing the convex hull means that a non-ambiguous and efficient representation of the required convex shape is constructed. The complexity of the corresponding algorithms is usually estimated in terms of  $n$ , the number of input points, and  $h$ , the number of points on the convex hull. In this work, we choose a very popular and efficient algorithm called *qhull* [108] to derive us the convex hull.

## 2.4 Data Structures for geometric objects

One of the most common representations of objects is the polygon/polyhedron mesh. A polygon/polyhedron mesh or unstructured grid is a collection of vertices, edges and faces that defines the shape of a polygon/polyhedron object in computer graphics. The faces usually consist of triangles, quadrilaterals or other simple convex polygons, since this simplifies rendering, but may also be composed of more general concave polygons/polyhedrons, or polygons/polyhedrons with holes.

Polygon/polyhedron meshes may be represented in a variety of ways, referring to the how the vertex, edge and face data are stored. The choice of the data structure is governed by the application, the performance required, size of the data, and the operations to be performed. For example, it's easier to deal with triangles than general polygons/polyhedron, especially in computational geometry. For certain operations it is necessary to have a fast access to topological information such as edges or neighboring



faces; this requires more complex structures such as the *winged-edge* representation. For hardware rendering, compact, simple structure is needed; thus the corner-table (triangle fan) is commonly incorporated into low-level rendering APIs such as DirectX and OpenGL.

The structure we use for the polygon/polyhedron model's representation is hierarchical. Every object is a list of surfaces, every surface is a list of polygons, and every polygon is a list of vertices.

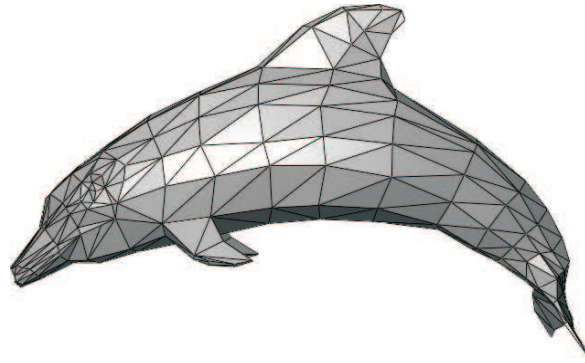


Figure 2.29: Example of a triangle mesh representing a dolphin.

# CHAPTER 3

## SKELETONIZATION

---

### 3.1 Skeleton Representation

### 3.2 Skeleton Extraction

### 3.3 Skeleton Refinement

---

In this chapter we focus on how to build a skeleton representing the meshes we wish to animate. A skeleton is a lower dimension object that represents the shape of the original object and capture to a certain level human visual perception. Since a skeleton has usually a simpler structure than the original object, many operations, e.g., shape recognition and deformation, can be performed more efficiently on the skeleton than on the full object. The process of generating such a skeleton is called *skeleton extraction* or *skeletonization*.

Skeletonization can be applied to arbitrary shape representations, such as polygonal models or parametric surfaces. In this work, without losing generality, we use triangulated models to represent 3D objects. Before we discuss about our framework, we will set some conditions on the input solid object:

- We consider manifold solids of genus 0 (zero cuttings along closed simple curves without rendering the resultant manifold disconnected).
- An effective shape decomposition method [50, 10, 61, 55] or an experience user must have partitioned the character mesh into visually meaningful or otherwise appropriate with regards to the application components.
- The distance between any two adjacent points of the solid model is under a certain threshold.

Finally, since the triangular representation does not provide us information about component connectivity, a description file similar to the BVH file format (without motion data) is provided with high level skeleton hierarchy information.

In this chapter, we propose a method that produces a refined skeletonization of a solid character. For a given polyhedron  $P$ , skeleton extraction methods constructs a visually satisfactory skeleton for the model from local skeletons (bones) extracted from each component of the character and from centroids of union points (joints) constructed from each pair of connecting components performing a depth-first traversal of the skeleton hierarchy tree. We study three interesting approaches, the Opening, the Centroid and the Principal Axis techniques (see e.g [63] for further information). Then, we modify them to derive better skeletal representations that are more appropriate for our application. Moreover, we refine the associated skeleton components by approximately aligning them with a set of vectors, and we, finally, connect the refined skeleton components to form a global skeleton of the input model.

Our proposed approach makes following technical contributions.

- The skeletonization method is independent of the character’s module size, it has no convexity requirements and is invariant under distortion and deformation of the input model.
- The refined skeleton extraction technique is applicable in animation, shape recognition, collision detection and automatic navigation.

### 3.1 Skeleton Representation

Information about the skeleton hierarchy structure tree is provided with a file with format similar to the BVH format. We do not use the BVH file because this file is a way to provide skeleton hierarchy information in addition to the motion data. Figure 3.1 provides an example of skeleton hierarchy file format and the corresponding skeleton hierarchy tree is shown in Figure 3.2.

The file has one part, a data section which describes the hierarchy of the skeleton and the constraints of the rotation angles of the skeleton’s joints. The format is nested. Each segment of the hierarchy contains some data relevant to that segment and it recursively determines its children. The start of the section the name of the root segment of the hierarchy to be defined followed by three pairs of rotations angles which define the joint’s valid rotation range. The order of the rotation angles appears, it goes X rotation range of angles, followed by the Y rotation range of angles and finally the Z rotation. The line following the root joint contains a single left curly brace “{”, the brace is lined up with the joint’s name. On the line of data following the “{”, can be one of two keywords, either you will find another name of a segment or you will see the ”*End Site*” keyword. A joint definition is identical to the root definition we describe above. This is where the recursion takes place, the rest of the parsing of the joint information proceeds just like a root. The end site information ends the recursion and indicates that the current segment is an end effector (has no children). The end of any joint, end site or root definition is denoted by

a right curly brace “}”. This curly brace is lined up with its corresponding right curly brace.

```

component_1 MIN_ANGLE_X MAX_ANGLE_X MIN_ANGLE_Y MAX_ANGLE_Y MIN_ANGLE_Z MAX_ANGLE_Z
{
  component_2 MIN_ANGLE_X MAX_ANGLE_X MIN_ANGLE_Y MAX_ANGLE_Y MIN_ANGLE_Z MAX_ANGLE_Z
  {
    component_3 MIN_ANGLE_X MAX_ANGLE_X MIN_ANGLE_Y MAX_ANGLE_Y MIN_ANGLE_Z MAX_ANGLE_Z
    {
      End Site
      {
      }
    }
    component_4 MIN_ANGLE_X MAX_ANGLE_X MIN_ANGLE_Y MAX_ANGLE_Y MIN_ANGLE_Z MAX_ANGLE_Z
    {
      End Site
      {
      }
    }
    .
    .
    .
  }
}

```

Figure 3.1: Skeleton representation file format.

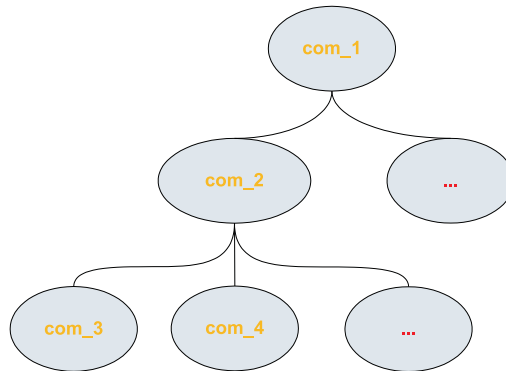


Figure 3.2: The produced skeleton hierarchy for Figure 3.1 example.

## 3.2 Skeleton Extraction

In this section we study the local skeletonization problem of a component. First, we examine a simple and fast approach, called here as the *Opening Method*, which usually produces inappropriate skeletons. The other two methods have been discussed at [63] and are the *Centroid* and *Principal Axis* methods. The Centroid Method is a simple approach but sometimes can lead to skeleton morphs that don't represent the shape of object. The other method which is based on the principal axis of component, is slightly more expensive to compute, but results in improved skeletons. We studied and furthermore modified them to derive better skeletal representations.

### 3.2.1 Opening Method

An opening consists of the common (joining) points of two adjacent components. Openings are created when a component is split into sub-components during a decomposition process. The easiest way to construct a skeleton of a component of a modular object is to connect the centroids of the openings, called *opening centroids*, to each other. Figure 3.3 shows the generated opening centroids between connecting neighbor components.

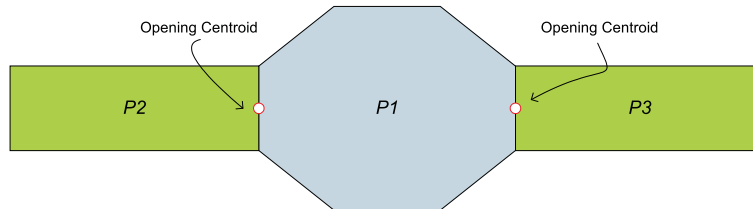


Figure 3.3: The generated opening centroids between P1 and the connecting neighbor components P2 and P3.

We arbitrarily pick the centroid of the root node of the object's skeleton tree as root point. The selected point is also the root point of the global skeleton. Then, we compute the opening centroids of all its children components and connect each of them with the centroid. For each child, we perform the same process considering as root point, the opening centroid where this component connect with its parent. Therefore, we connect the root point with the opening centroids of all its children components. Finally, this process ends when we reach the leaves of the skeleton tree.

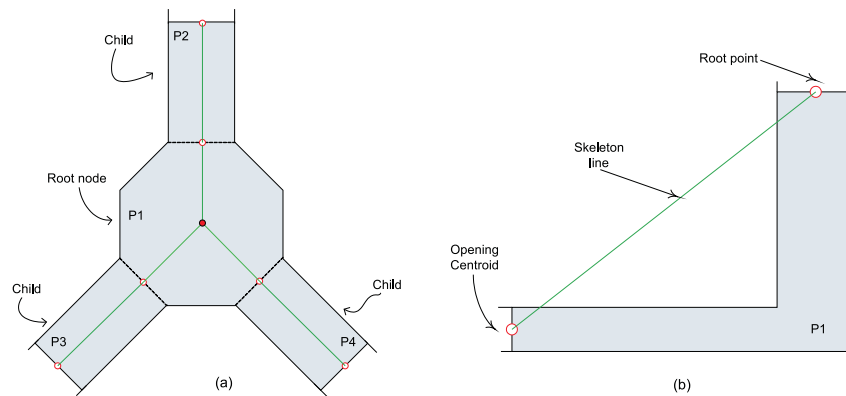


Figure 3.4: (a) A simple example of a skeletal representation using the Opening Method. (b) An example that illustrates the inappropriate skeleton representation that arises when skeletonization is based only on opening centroids.

This approach is straight forward and fast but frequently generates inappropriate results. The major drawback of this approach is that it may produce skeleton segments that intersect the component (Figure 3.4b) and skeletons that do not capture accurately the shape of the object.

### 3.2.2 Centroid Method

The following method is more advanced method than the first one since it uses an additional feature to extract skeleton segments. The aim of this method is to construct skeleton segment using the center of the mass of the component in addition to the opening centroids. Sometimes, however, the centroid may lie outside the component producing erroneous skeletons. To overcome this shortcoming we use the centroid of its kernel on *star-shaped* components (polyhedra with non-empty kernel). In non star-shaped components the center of their mass is used instead. Figure 3.5(a) shows the resulting skeletal representation of the example of section 3.2.1. Moreover, Figure 3.5(b) illustrates the centroid method effectiveness in the example of Figure 3.4(b).

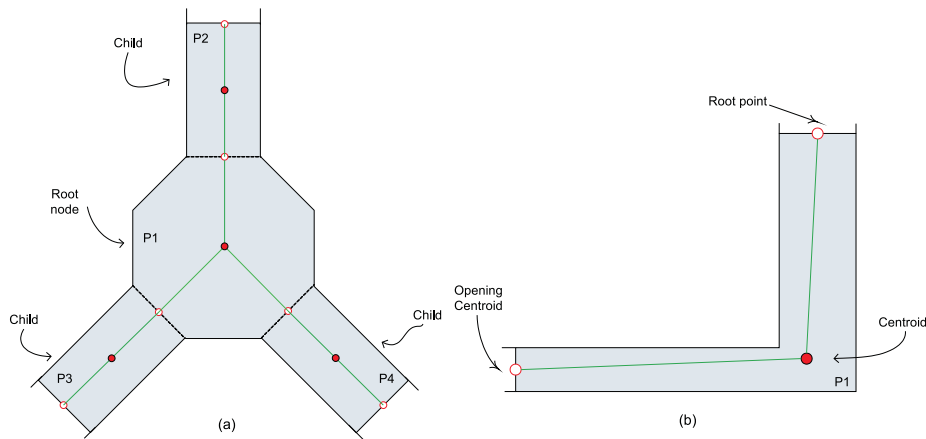


Figure 3.5: (a) A simple example of a skeletal representation using the Centroid Method. (b) The Opening Method problem in 3.4b addressed using the Centroid Method.

Knowing the skeleton hierarchy information, we start from the root node and we compute the centroid of component's kernel. The selected point is also the root point of the final skeleton. Then, we compute the opening centroid of all its children components and connect each of them with kernel's centroid. Likewise, for each child we perform the same process considering as root point the opening centroid where this component connects with its parent. Finally, we connect the root point with kernel's centroid and kernel's centroid with opening centroids of all their children components. Finally, this process ends when we reach the leaves of the skeleton tree.

Although this approach is efficient and produces good results in simple cases, one of the major drawbacks of this skeletonization method is its inability to represent some types of shapes. For example, the skeleton of a cross-like model in Figure 3.6 extracted using its centroids is only a segment instead of two crossing segments.

### 3.2.3 Principal Axis Method

In [63] proposed to overcome failures of previous methods by extracting a skeleton from a component by connecting the opening centroids to the principal axis segment that lies within the convex hull of the component and goes through the center of its mass.

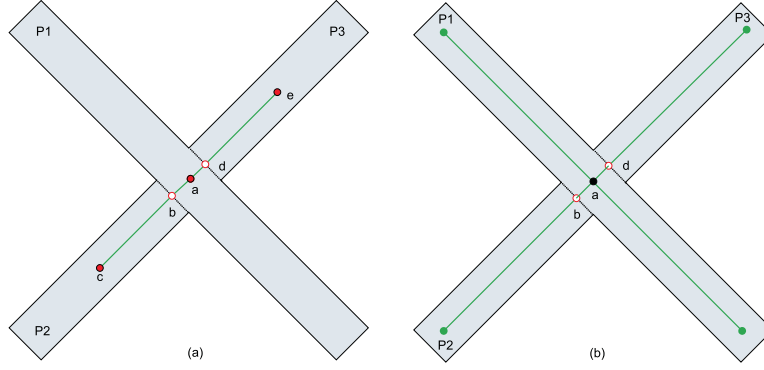


Figure 3.6: (a) An example that illustrates the problem that arises when skeletonization is based only on centroids. (b) This problem can be solved using the principal axis. Points  $b$  and  $d$  are the centers of the openings and  $a$ ,  $c$  and  $e$  the kernel's centers of the components  $P1$ ,  $P2$  and  $P3$ , respectively.

We denote this principal axis segment as  $PA_{MC}(CH)$ . To achieve this,  $PA_{MC}(CH)$  is segmented in  $k + 1$  line segments of equal length by picking  $k$  link points on  $PA_{MC}(CH)$ , where  $k$  depends on the minimum skeleton linkage length. This method maps the opening centroids on the  $PA_{MC}(CH)$  by minimizing the total interconnection length and the number of link points used. This is reduced to an optimization matching problem. The final skeleton of the component contains line segments that connect opening centroids to link points and line segments that interconnect link points.

Our approach adapts this skeleton extraction technique for use in our animation framework (see Algorithm 1). Let  $OC(C)$  be the set of the opening centroids  $oc_i$  of component  $C$ ,  $OC(C) = \{oc_1, \dots, oc_n\}$  and  $N_{OC}$  the cardinality of  $OC(C)$ . First, we select the principal axis segment of the component's convex hull that resides within the interior of the component. For even better results we have used as major axis an axis parallel to the principal axis of the convex hull that goes through the kernel centroid of the component to ensure that skeleton segments have the least possible intersection with the component's boundary. We denote this major axis segment as  $PA_{KC}(C)$ . Moreover, the cardinality of the set of link points to which the opening centroids connect varies from 1 to  $N_{OC}$ . We subdivide the  $PA_{KC}(C)$  in a number of uneven segments by picking as link points the projections of the opening centroids on  $PA_{KC}(C)$  (see Figure 3.7). Let  $P_{PA_{KC}}(oc_i)$  be the projection of opening centroid  $oc_i$  on  $PA_{KC}(C)$ . If this projection lies outside  $PA_{KC}(C)$  then we use the closest end point of  $PA_{KC}(C)$ . We then sort opening centroids according to their closest points by observing that two centroids are likely to be grouped when their closest points in  $PA_{KC}(C)$  are close.

Our matching algorithm uses a dynamic programming concept, enhanced with new score functions which aim at minimizing the total length of the mapping and the number of the mapped points and maximizing the length of the utilized  $PA_{KC}(C)$ . Details of the *grouping algorithm* are discussed in Section 3.2.3.

After grouping has been performed, we create the set of skeleton edges. Beyond the

standard connections between the opening centroids and the link points, we use extra skeleton edges based on the mapping result. The *connecting algorithm* is discussed in detail in Section 3.2.3.

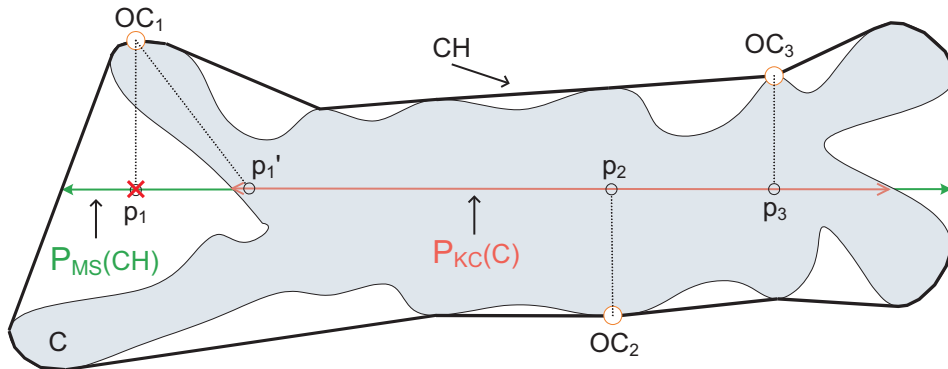


Figure 3.7: Difference between  $PA_{MC}(CH)$  and  $PA_{KC}(C)$ . The projection of opening centroid  $oc_2$  ( $p_1$ ) lies outside  $PA_{KC}(C)$ , so it is replaced by the closest end point of  $PA_{KC}(C)$ , this is ( $p_1'$ ).  $p_2$  and  $p_3$  points are the projections of opening centroids  $oc_2$  and  $oc_3$  on  $PA_{KC}(C)$ , respectively.

Further, the covariance-aligned principal directions approximation algorithm will not derive the optimal orientations but can provide an initial direction which if combined with qualitative characteristics may improve the initial solution considerably. Our approach improves the principal axis orientation through slight modifications using two individual processes, the first is based on local features and the second uses knowledge inherited from the component hierarchy.

Finally, the basic method depends only on the principal axis ignoring the other two principal directions of the component. Thus, we propose to add four more joints which represent the two ends of the other axes segments making the skeletal representation more topologically expressive. (this is obsolete for character skinning but may be used effectively in subsequent steps of the animation pipeline). Figures 3.8(a) and 3.8(b) illustrate the difference of skeleton representations using this variation.

The principal axis algorithm is more expensive to compute than the other two methods due to the principal axis and kernel centroid computation but obtains a higher quality skeleton by capturing the topology of the character's shape more accurately.

We start from the root node of the skeleton tree and we calculate the opening centroids of the component with all its children. In addition, we calculate the segment of principal axis which resides in the interior of the component. We further sort these opening centroids with respect to the closest points (projections) in the produced line and we execute the grouping algorithm. Then, we use the connecting algorithm for creating the skeleton segments. If the grouping algorithm returns one group (which means that we have one point on the principal axis) then apparently we choose this point to be the component's root point. On the other hand, we choose as root point the point which is the middle of the returned points. For each child component, we perform the same process considering



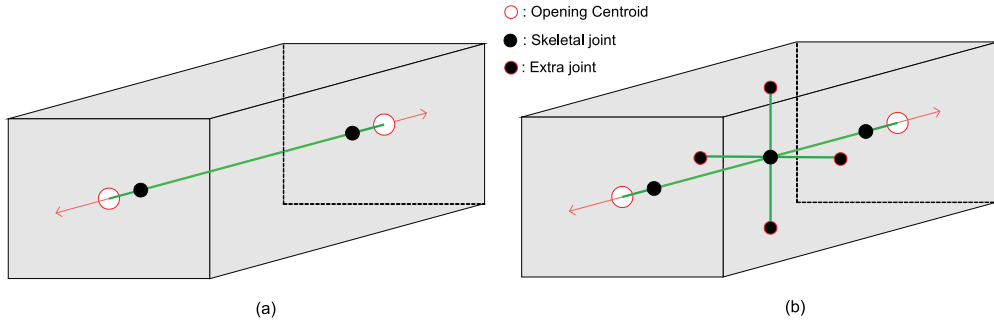


Figure 3.8: Skeletonization using (a) only PA, (b) all principal axes.

as root point the opening centroid where this component connects with its parent. Finally, this process ends when we reach the leaves of the skeleton tree.

The principal axis algorithm is obviously more expensive than the other two methods to compute due to the covariance-principal axis computation but carries out a first-rate skeleton quality by encoding the topology of the model's shape efficiently.

---

**Algorithm 1** `principal_axis( $C$ )`

---

1.  $OC := \text{computeOC}(C)$ ;
  2.  $KC := \text{computeKC}(CH(C))$ ;
  3.  $PA_{KC} := \text{computePA}(C, KC, CH(C))$ ;
  4. **for each**  $oc_i \in OC$  **do**
  5.      $P_{PA_{KC}}[oc_i] := \text{project}(oc_i, PA_{KC})$ ;
  6.  $\text{sort}(OC, \{P_{PA_{KC}}[oc_i]\})$ ;
  7.  $\text{group}(OC, PA_{KC})$ ;
  8.  $\text{connect}(OC, PA_{KC})$ ;
- 

### Principal Axis Approximation

S. Gottschalk in [40] proposed an efficient and accurate algorithm to approximate the collection of polygons with an oriented bounding box (OBB) of similar dimensions and orientation using the covariance matrix. For long and thin objects, an OBB axis should be aligned with the direction of the objects and for a flat object, with the normal of the flat object. These directions correspond to the principal directions of the objects, and the principal component analysis method discussed in Appendix C can be used here. So, we can find the principal axis of a component by computing the OBB axes of its convex hull. The proposed solution is to use a continuous formulation of covariance, computing the covariance across the entire face of the primitives. Because the convex hull must be computed, the algorithm takes  $O(n \log n)$ . Given  $n$  triangles  $(p_k, q_k, r_k)$ ,  $0 \leq k \leq n$ , in the convex hull, the covariance matrix is given by

$$C_{ij} = \left( \frac{1}{12A_H} \sum_{k=0}^n (a_k (9m_{k,i}m_{k,j} + p_{k,i}p_{k,j} + q_{k,i}q_{k,j} + r_{k,i}r_{k,j})) \right) - m_{H,i}m_{H,j}$$

where  $a_k = |(q_k - p_k) \times (r_k - p_k)|/2$  is the area and  $m_k = (p_k + q_k + r_k)/3$  is the centroid of the triangle  $k$ . The total area of the convex hull is given by:  $A_H = \sum a_k$ , and the centroid of the convex hull,  $m_H = \frac{\sum a_k m_k}{A_H}$ , is computed as the mean of the triangle centroids weighted by their area.

The three eigenvectors of  $C$  will be mutually orthogonal. After normalization, the three eigenvectors become axes of OBB. The Singular Value Decomposition method discussed in Appendix B.1 can be used here to extract these axes, which in turn can then serve as the OBB's orientation matrix. The principal axis is the eigenvector of the covariance matrix which corresponds to the largest eigenvalue.

Finally, we must find the maximum and minimum extends of the original triangle set along each axis in order to size the OBB since as we will discuss later, OBBs is necessary at the deformation system. This can be done by searching the two maximum distances  $m_1$  and  $m_2$  from the centroid to the boundary of the convex hull across each axis and its negative. Then, we must only set the half length of each axis as the  $(m_1 + m_2)/2$ , and finally, re-compute the position of the centroid of the OBB.

## Grouping Algorithm

Our approach manages to minimize the total mapping length and the cardinality of the set of link points, while at the same time maximizes the used  $PA_{KC}(C)$  length. Further, it does not depend on system determined constants, thus being widely applicable. Let  $d(\vec{p}_1, \vec{p}_2)$  be the euclidean distance between point  $\vec{p}_1$  and point  $\vec{p}_2$ . For a set of opening centroids of a component  $C$ ,  $oc_{ij}(C) = \langle oc_i, \dots, oc_j \rangle$ , we compute their link points on  $PA_{KC}(C)$  as the average of their projections by

$$L_{PA_{KC}(C)}(oc_{ij}(C)) = \frac{\sum_{k=i}^j P_{PA_{KC}(C)}[oc_k]}{|oc_{ij}(C)|} \quad (3.1)$$

$\forall oc_k \in oc_{ij}(C)$ , we evaluate its normalized variation as

$$V(oc_k) = \frac{d(oc_k, L_{PA_{KC}(C)}(oc_{ij}(C))) - d_{min}(oc_k)}{\max_{1 \leq l \leq N_{OC}} \{d(oc_k, oc_l)\} - d_{min}(oc_k)} \quad (3.2)$$

where  $d_{min}(oc_k) = d(oc_k, P_{PA_{KC}(C)}[oc_k])$ .

Moreover, we define the ratio of the  $PA_{KC}(C)$  length which is vanished after grouping as the  $PA_{KC}(C)$  length which is generated among the projections of these opening centroids divided by the maximum used  $PA_{KC}(C)$  length.

**Definition 3.1.** The *normalized merging score function*  $F_1$  for  $OC_{ij}(C)$  group set is defined as the average of the total distance cost and the ratio of the not used  $PA_{KC}(C)$

length,

$$F_1 = \frac{\frac{\sum_{k=i}^j V_k}{|OC_{ij}(C)|} + \frac{d(P_{PA_{KC}(C)}[oc_i], P_{PA_{KC}(C)}[oc_j])}{d(P_{PA_{KC}(C)}[oc_1], P_{PA_{KC}(C)}[NOC])}}{2} \quad (3.3)$$

**Definition 3.2.** The *normalized separating score function*  $F_2$  for groups  $G_x = \langle oc_i^x, \dots, oc_l^x \rangle$  and  $G_y = \langle oc_{l+1}^y, \dots, oc_j^y \rangle$  is defined as the average of the sums of their total distance cost and the complement of the ratio of  $PA_{KC}(C)$  length which is generated between these groups,

$$F_2 = \frac{\frac{\sum V(oc_k^x)}{|G_x|} + \frac{\sum V(oc_k^y)}{|G_y|} + (1 - gR_{PA_{KC}(C)})}{3} \quad (3.4)$$

where

$$gR_{PA_{KC}(C)} = \frac{d(P_{PA_{KC}(C)}[oc_l^x], P_{PA_{KC}(C)}[oc_{l+1}^y])}{d(P_{PA_{KC}(C)}[oc_1], P_{PA_{KC}(C)}[NOC])} \quad (3.5)$$

In algorithm 2, we use  $G[i, j]$  and  $G_i \cup G_j$  to denote the optimal solution for the sub-problem  $\langle oc_i, \dots, oc_j \rangle$  and the joint of two groups  $G_i$  and  $G_j$  without merging them to one group, respectively.

---

**Algorithm 2** group( $OC, PA_{KC}$ )

---

1. **for each**  $oc_i \in OC$  **do**  $G[i, i] = oc_i$ ;
  2. **for**  $l = 2$  **to**  $N_{OC}$  **do**
  3.     **for**  $i = 1$  **to**  $N_{OC} - l + 1$  **do**
  4.          $j = i + l - 1$ ;
  5.          $G[i, j] = \langle oc_i, \dots, oc_j \rangle$ ;
  6.          $s_1 = F_1(G[i, j], PA_{KC})$ ;
  7.         **for**  $k = 1$  **to**  $j - l$  **do**
  8.              $s_2 = F_2(G[i, k], G[k + 1, j], PA_{KC})$ ;
  9.             **if**  $s_2 < s_1$  **then**
  10.                  $G[i, j] = G[i, k] \cup G[k + 1, j]$ ;
  11.                  $s_1 = s_2$ ;
  12. **return**  $G[0, N_{OC}]$ ;
- 

## Connecting Algorithm

Once we know the optimal mapping, we connect the opening centroids to the matched link points. Opening centroid grouping often generates skeletons that do not capture important topological information. Therefore, the connection algorithm works with the following rules:

1. If all opening centroids are grouped to one link point and this point is close to:

- $PA_{KC}(C)$ 's centroid, we connect it with the  $PA_{KC}(C)$ 's end points on both sides of its centroid.
  - one of  $PA_{KC}(C)$ 's end points, we connect it with the  $PA_{KC}(C)$ 's end point on the other side of  $PA_{KC}(C)$ 's centroid.
2. If opening centroids are connecting to more than one link point and these points are close to:
- $PA_{KC}(C)$ 's centroid, we connect the first and the last link points with the  $PA_{KC}(C)$ 's end points on both sides of  $PA_{KC}(C)$ 's centroid, respectively.
  - one of  $PA_{KC}(C)$ 's end points, we connect the first or the last link point with the  $PA_{KC}(C)$ 's end point on the other side of  $PA_{KC}(C)$ 's centroid.

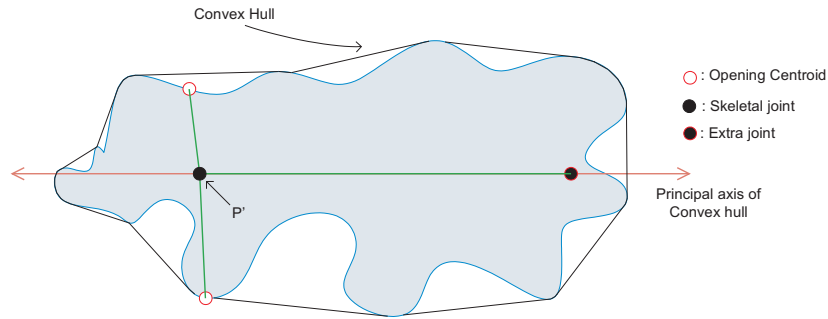


Figure 3.9: If all opening centroids connect to one point which is close to one end of the  $PA_{KC}$  then we connect this point with the  $PA_{KC}$  end point on the other side of the component.

### 3.3 Skeleton Refinement

As covariance-aligned OBBs are not optimal due to the approximation computation process, it is reasonable to suspect that could be improved through slight modifications. There are two major problems of the PA covariance-computation. There is high possibility that either the eigenvector of the covariance matrix which corresponds to the largest eigenvalue is not the principal axis or the selected principal axis has not the accurate orientation we wanted. Certainly, the covariance-aligned OBBs algorithm will not derive the perfect principal axis orientation but can provide basic information that if it is combined with local and parent features that define a qualitative orientation, may improve the initial solution efficiently. Our approach perfects the axes of the oriented bounding boxes orientation using two individual processes, the first one uses local features and the second one uses knowledge given from its parent node. Figure 3.13 illustrates the qualitative superiority of our refined skeletons and aligned OBBs over the original principal axis algorithm.

### 3.3.1 Local Refinement Method

To achieve optimal or satisfactory orientation results we use *Local Refinement* to approximately align principal directions along some qualitative features through slight modification. We define these features as the extracted skeletons segments from Centroid method; vectors created by connecting opening centroids with kernel's centroid. The maximum rotation angle is user determined parameter and we set it less than  $20^\circ$  degrees based on our own experience. Changes that demand rotations by large angles are rejected. The proposed algorithm performs *weighted* vector alignment for each opening centroid so that each execution will not undo previous improvements. If the component has more than one opening centroids, we align with regards to the closest principal direction by  $angle = \frac{A}{weight}$ , where  $A$  is the angle between the given vector and its closest axis and  $weight = \frac{1}{N_{OC}+1}$  except from the opening centroid which lies at the root. Considering that this has more importance to the overall refinement, we weighted by  $2 * weight$ , such that  $\sum_{i=1}^{N_{OC}} weight_i = 1$ . Closest is the principal direction which has the minimum angle between itself and the given vector. When component has only one opening centroid, we fit its corresponding vector with its closest principal direction. If the closest principal direction is different from principal axis then the latter will be the selected axis. Figures 3.10 and 3.11 illustrate local refinements of a root and not-root components, respectively.

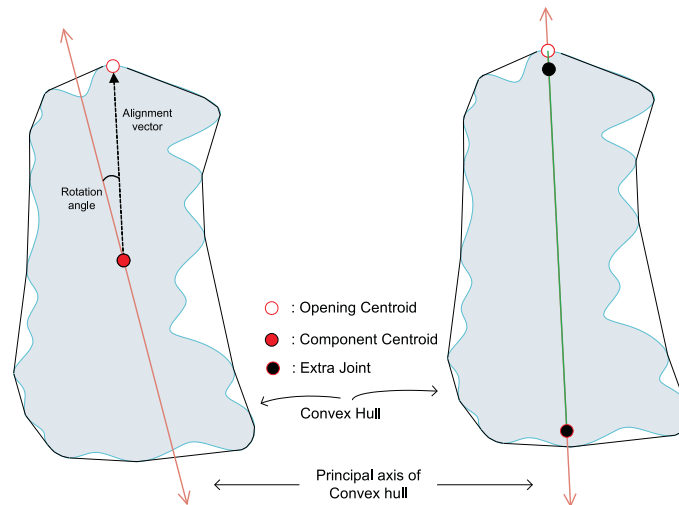


Figure 3.10: This example shows a rotation refinement on a component with one opening centroid. (Left) shows the initial principal axis orientation and (right) is illustrated the produced skeleton after alignment execution.

Let symbol  $\angle(\vec{p}_1, \vec{p}_2)$  be the angle between  $\vec{p}_1$  and  $\vec{p}_2$  vectors. In algorithm 3, we use  $PD = \langle PD_0, PD_1, PD_2 \rangle$  and  $PD_j^*$  to denote component's principal directions and principal axis, respectively. The maximum rotation angle and the component's root are denoted respectively as  $maxAngle$  and  $C_{root}$ .

---

**Algorithm 3** local\_refine( $C, OC, KC, PD, Root$ )

---

- 1:  $w = N_{OC} + 1$ ;
- 2: **if**  $C \neq Root$  **then** align( $C_{root} - KC, 2w, PD$ );
- 3: **For each**  $oc_i \in OC$  **do** align( $oc_i - KC, w, PD$ );

**Function** align( $Vector, Weight, PD$ )

- 1: **for**  $i = 0$  **to**  $2$  **do**  $A_i = \angle(PD_i, Vector)$ ;
  - 2:  $A_K = \max_{0 \leq i \leq 2} \{A_i\}$ ;  $K = \{PD_k \mid A_k = A_K\}$ ;
  - 3:  $A_K = A_K / Weight$ ;
  - 4: **if**  $A_K > maxAngle$  **then** return;
  - 5: rotate( $PD, A_K$ );
  - 6: **if**  $PD_j^* \neq K$  **then**  $PD_j^* = K$ ;
- 

### 3.3.2 Parent Refinement Method

We extend the orientation improvements performed individually on each component by performing optimal fitting of the local principal directions of neighbor components. The algorithm starts from the children of the root node of the skeleton tree performing the same process to their children until it reaches the tree leaves. Its limitation is the dependence on root component principal directions orientations result. For each component, the *Parent Refinement* process first aligns with regards to the parent's principal direction which is closer to the child's principal axis. Subsequently, from the rest of the child's and parent's principal directions we detect those that are closer and aligns them. Note that fitting two principal directions from different components, results in rotating the other principal directions too. In Figure 3.12 we can notice the topological quality improvement of skeleton using this method.

In algorithm 4, we use  $PD^C = \langle PD_0^C, PD_1^C, PD_2^C \rangle$  and  $PD^P = \langle PD_0^P, PD_1^P, PD_2^P \rangle$  to denote respectively child's and parent's principal directions. Finally,  $PD_j^{C,*}$  defines the child's principal axis.

---

**Algorithm 4** parent\_refine( $PD^C, PD^P$ )

---

- 1: **for**  $i = 0$  **to**  $2$  **do**  $A_i = \angle(PD_i^P, PD_j^{C,*})$ ;
  - 2:  $A_K = \max_{0 \leq i \leq 2} \{A_i\}$ ;  $K^P = \{PD_k^P \mid A_k = A_K\}$ ;
  - 3: rotate( $PD^P, A_K$ );
  - 4:  $A = \min_{0 \leq i \leq 2} \{ \angle(PD_i^C, PD_i^P) \mid PD_i^C \neq PD_j^{C,*} \}$   
**and**  $PD_i^P \neq K^P$ };
  - 5: rotate( $PD^C, A$ );
-

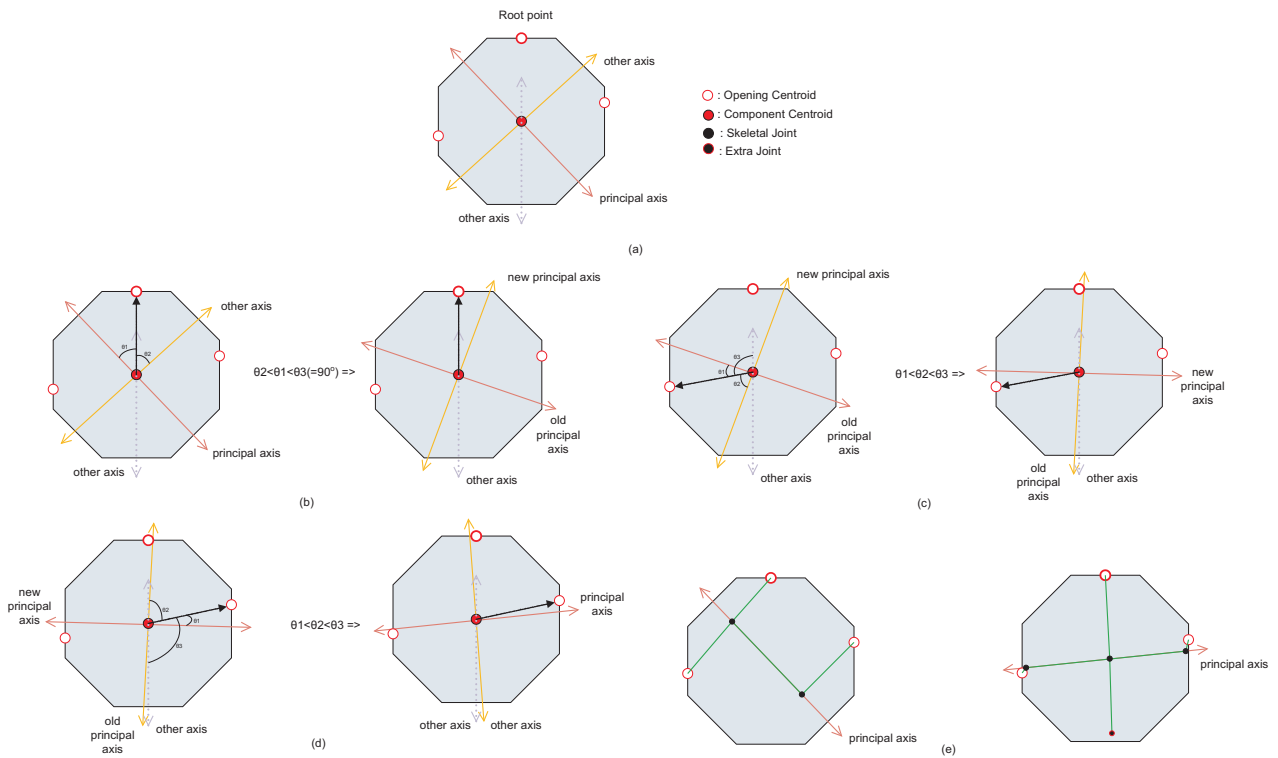


Figure 3.11: (a) The covariance-computed principal axes of the component. (b) The first refinement execution: The yellow axis is closer to the given vector, so it will be the new principal axis. (c) The second refinement execution: Again, the yellow axis is closer to the given vector. (d) The last refinement execution: The pink axis is closer to the given vector, so it will be the new principal axis (The initial and the new principal axes are now the same). (e) The difference of the extracted skeletons, before and after the refinement.

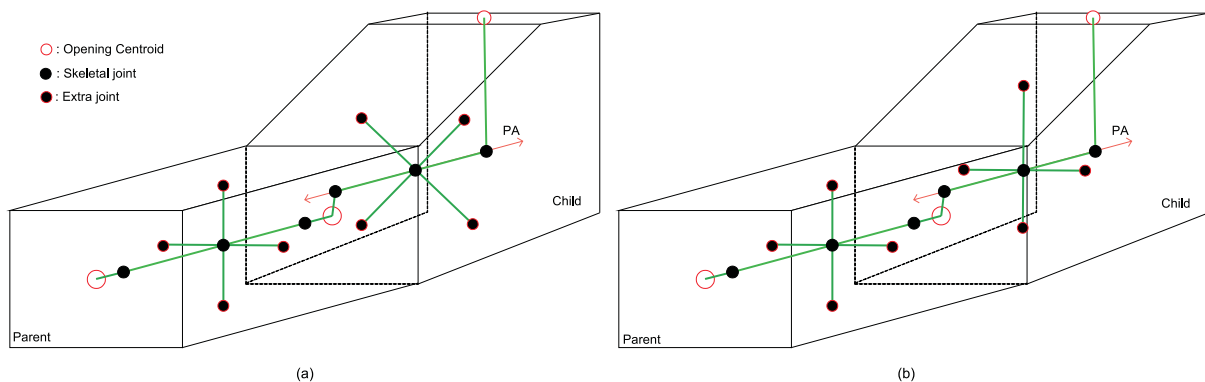


Figure 3.12: Skeleton extraction without (a) and with (b) using the Parent Refinement Algorithm.

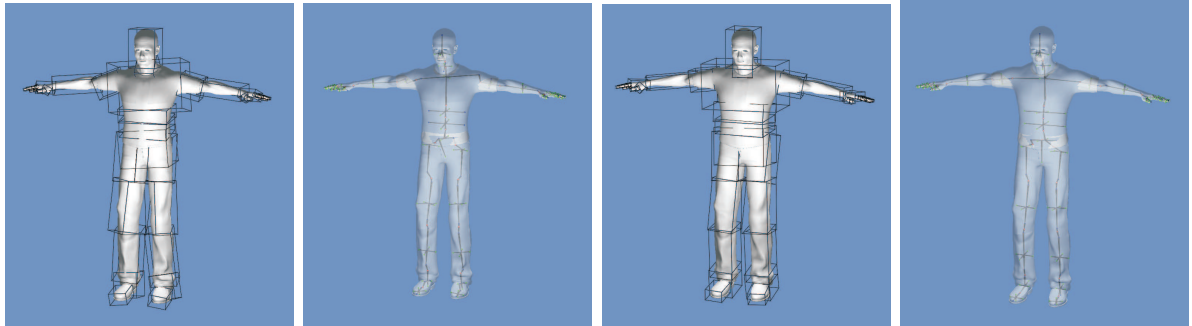


Figure 3.13: Oriented bounding boxes and skeletons of “Human” components: (left) Original configuration (right) After alignment.



# CHAPTER 4

## ARTICULATED ANIMATION

---

4.1 Representing Articulated Figures

4.2 Keyframing Animation System

4.2 Forward Kinematics Skinning System

4.2 Gap Reconstruction Process

---

In the simplest form of computer animation we use a standard renderer to produce consecutive frames, wherein the animation consists of relative movement between rigid (articulated) bodies and possibly movement of the view point or virtual camera.

In this work, we propose an integrated framework for robust skeletal animation of articulated modular solid objects which is illustrated in Figure 4.1. First, a keyframing animation system evaluates the rotational joint angles of the extracted skeleton for every frame. Then, these values are used to examine if this motion should be rejected by ensuring that they are consistent with the joint's angle constraints and detecting collision between the moving and the static object's modules. If the motion is satisfactory under these constraints then the forward kinematics rigid skinning system performs the specified animation using the skeleton representation model. Finally, although it is simple, rigid skinning makes it difficult to obtain sufficiently smooth skin deformation in areas near the joints, so we introduce a new system which aims to solve this appearance problem efficiently.

This chapter begins by examining the representation alternatives of an articulated animation. In the following sections, the sub-systems of our skeletal animation framework, the keyframing animation system, the forward kinematics skinning system, and the gap construction system are described analytically.

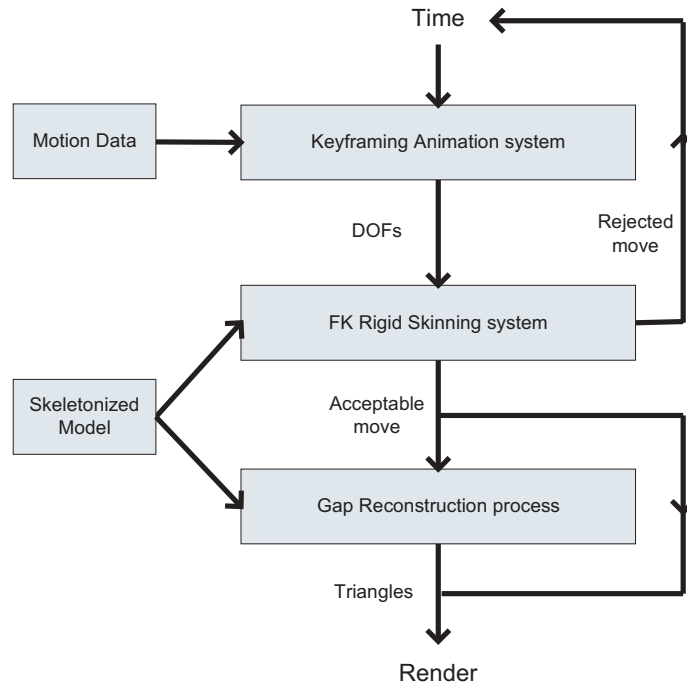


Figure 4.1: A framework for skeletal animation of articulated figures.

## 4.1 Representing Articulated Figures

The first problem in articulated animation is how to represent an articulated figure.

The *DH* notation [30] is an economical relative system, where each coordinate frame is specified with respect to the previous. However, Sims and Zeltzer [79] point out that this notation is only really suitable for manipulators that consist of a single chain with one end fixed, and cannot incorporate branching joints and skeletal components. Moreover, they introduce a less economical but more intuitive system called *axis-position (AP)* joint representation storing:

1. The position of the joint
2. The orientation of the axis of the joint
3. Pointers to the skeletal components that each joint is attached to

This notation requires seven parameters (three for position, three for axis orientation and one for joint angle) compared with the four values in the *DH* notation.

## 4.2 Keyframing Animation System

This section refers to the various techniques in which the animator is responsible for specifying most of the information. That is, the animator is working at a fairly low level of abstraction. These techniques differ from model-based techniques in that there is a more direct relationship between the information provided by the animator and the resulting motion. In contrast, in model-based animation, the animator works at a high level of animation and, as a result, gives up more control of the specific motion, but less information is required from the animator.

Natural motion rarely happens along straight lines, so this should generally be avoided in animation and *arcs* should be used instead. Similarly, no real-world motion can instantly change its speed; this would require an infinite amount of force to be applied to an object. It is desirable to avoid such situations, in animations as well. In particular, the motion should start and end gradually (*slow in and out*). While hand-drawn animation is sometimes done via straight-ahead action with an animator starting at the first frame and drawing one frame after another in sequence until the end, *pose-to-pose action*, also known as *keyframing*, is much more suitable for computer animation. In this technique, animation is carefully planned through a series of relatively sparsely spaced key frames with the rest of the animation (in-between frames) filled in only after the keys are set. This allows more precise timing and allows the computer to take over the most tedious part of the process using algorithms presented in the next section.

The term keyframing can be misleading when applied to 3D computer animation since no actual completed frames are typically involved. At any given moment, a 3D scene being animated is specified by a set of numbers: the positions of joints, their RGB colors, modeling transformations, camera and light position and orientation, etc. These independent variables which are necessary to specify the state of a scene called *degrees of freedom (DOF)* of the scene. Changing the DOF values over time results in the animation of the figure. Lastly, it's useful to mark down that it will be nice to be able to limit a DOF to some range to prevent infeasible motions. Usually, in a realistic character, all DOFs will be limited except the ones controlling the root (for example, the elbow component of a human model could be limited from  $0^\circ$  to  $150^\circ$  degrees).

The vector space of all possible configurations of an articulated figure forms the basis of the *pose space* of the figure as the set of DOF defining the position, orientation and rotations of all joints constituting the figure. We can say more formally that the figure in a particular configuration is described by the pose vector:  $\Phi = (\varphi_1, \varphi_2, \dots, \varphi_N)$ . The dimension of the pose space in general is equal to the DOF of the articulated structure. For example, any unconstrained rigid body has six DOF, three translational and three rotational. Thus its pose vector is:  $\Phi = (x, y, z, r_x, r_y, r_z)$ . But in our case, the pose vector reduces to three rotational angles since the joint is *Ball and Socket* type as shown in Figure 4.2. However, when we want an interactive figure to move around through a complex environment, we need something to be responsible for the overall placement of the figure. We can solve this by using an additional extra channel data (3 translations, 3 rotations) called *character mover*. An animation can be thought of as a point moving

through pose space, or alternately as a fixed curve in pose space:  $\Phi = \Phi(t)$ . Generally, we think of an individual animation as being a continuous curve, but there's no strict reason why we couldn't have discontinuities (cuts). In these terms finding an animation reduces to finding an N-dimensional path in its pose space.

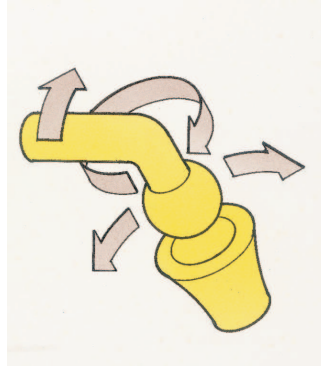


Figure 4.2: Ball and Socket joint.

Since the entire animation is an N-dimensional curve in pose space, we can separate that into N-dimensional curves, one for each DOF calling them *channels*:  $\varphi_i = \varphi_i(t)$ . A channel stores the value of a scalar function over some 1D domain (either finite or infinite) and will refer to pre-recorded or pre-animated data for a DOF, and not to the more general case of a DOF changing over time which includes physics, procedural animation, etc. A channel can be stored as a sequence of *key frames*. Key frames  $t_k$  are some number of important moments in time for each of the DOF which are selected to set values of this DOF (*key values*  $f_k$ ). Of course, we can directly set these values at every frame, but this will not be particularly efficient. Furthermore, the closer key frames are to each other, the more control the animator has over the result; however the cost of doing more work of setting the keys has to be assessed. It is logical, therefore, typical to have large spacing between keys in parts of the animation which are relatively simple, concentrating them in intervals where complex action occurs. Once the animator sets the keys  $(t_k, f_k)$ , the system has to compute dynamically values of  $f$  for every possible key frame  $t$ . This process is called *curve fitting*, as it involves finding curves that fit the data reasonably well. Since we are not ultimately interested only in a discrete set of values, it is convenient to treat this as a classical 1-D interpolation problem which fits a continuous animation curve  $f(t)$  through a provided set of data points (Figure 4.3).

Since the animator provides only the keys and not the derivative (tangent), method which compute all necessary information directly from keys are preferable for animation. The speed of parameter change along the curve is given by the derivative of the curve with respect to time  $\partial f / \partial t$ . Therefore, to avoid sudden jumps in velocity,  $C^1$  continuity is typically necessary. Key frames are usually drawn so that the incoming tangent points to the left (earlier in time) as shown in Figure 4.3. The arrow drawn is just for visual representation and it should be remembered that if the two arrows are exactly opposite,

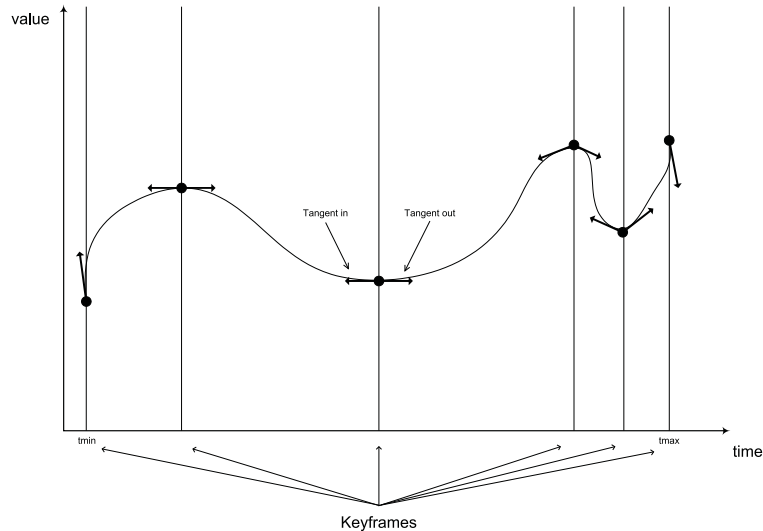


Figure 4.3: A continuous curve is fit through the keys provided by the animator.

that actually means the tangents are the same. A higher degree of continuity is typically not required from animation curves, since the second derivative, which corresponds to acceleration or applied force, can experience very sudden changes in real-world situations, and higher derivatives do not directly correspond to any parameters of physical motion. One of the best choices for defining the curves of the individual *spans* between the keys is by 1-D interpolation usually piecewise cubic Hermite curves.

As we discussed above, rather than store explicit numbers for tangents, it is often more convenient to store a *rule* and re-compute the actual tangent as necessary. Usually, separate rules are stored for the incoming and outgoing tangents. Common rules for Hermite tangents are shown in Figure 4.4 and include:

**Flat** : equals to zero and is particularly useful for making “slow in” and “slow out” motions (acceleration from a stop and deceleration to a stop)

**Linear** : points to next/last key and is particularly used on the first or last tangent respectively.

**Smooth** : automatically adjust tangent for smooth results and is particularly used on all keys except the first and last tangents.

The two main setup computations a keyframe channel needs to perform are:

- Compute the tangents values from the stored rules (Figure 4.4).
- Compute the Hermite cubic curves coefficients from the key values and the computed tangents (see Hermite at subsection 2.1.3).

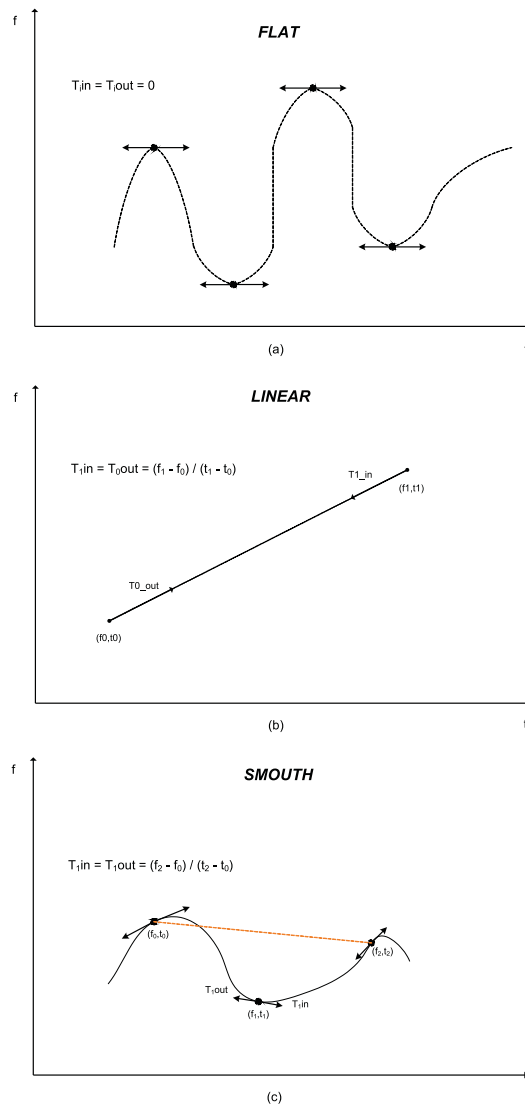


Figure 4.4: Form and evaluation of (a) flat, (b) linear, and (c) smooth Hermite tangents rules.

Finally, keyframe channels often specify *extrapolation modes* to determine how the curve is extrapolated before the first and after the last key frames. Usually, separate extrapolation modes can be set for before and after the actual data. We have to note that extrapolation applies to the entire channel and not to individual keys frames. Common extrapolation choices are shown in Figure 4.5 and include:

- Constant value** : hold first/last key value
- Linear** : use tangent at first/last key
- Cyclic** : repeat the entire channel
- Cyclic Offset** : repeat with value offset
- Bounce** : repeat alternating backwards and forwards

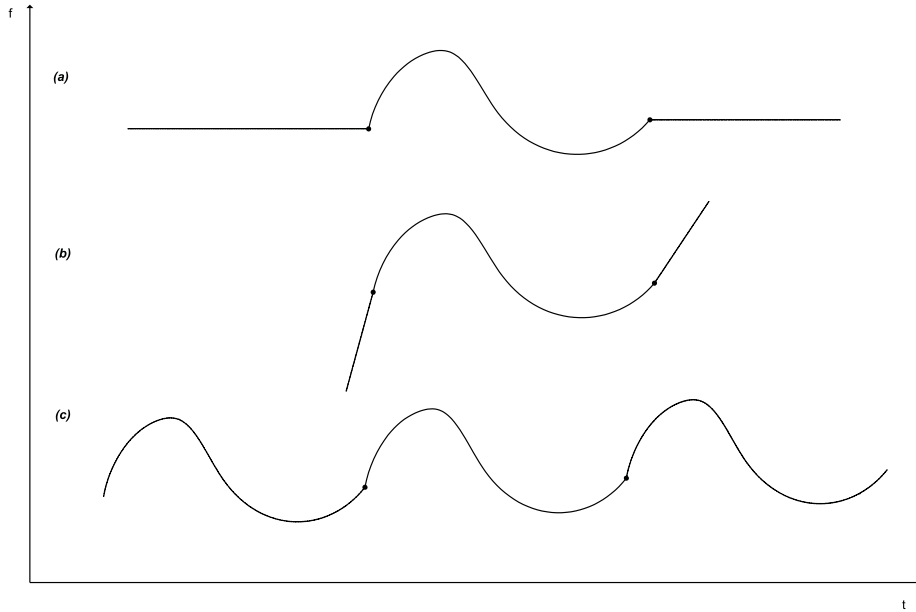


Figure 4.5: (a) Constant value, (b) linear, and (c) cyclic channel extrapolation modes.

Once we have defined the setup computation functions of the keyframe channels, we will discuss now how the evaluation process must be implemented. The evaluation function needs to be very efficient, as it called many times while playing animations. The procedure consists of two main components:

- Find the proper span on the curve.
- Compute the key value from the cubic equation for the chosen span.

As the key frames are irregularly spaced, we have to select the ones that are more appropriate for our application. If the key frames are stored as a linked list, we should walk through the entire list looking for the appropriate span. If we maintain the keys in an array, we can use a binary search. This requires  $O(\log N)$  access time with no additional storage cost. For even faster access, one could use hashing algorithms, but that is probably not necessary, as they require additional storage and most real channel accesses can take advantage of coherence. But, since the access to the channel data is sequential then doing a binary search for each channel evaluation for each frame is not efficient. The high performance evaluation solution is to loop through the keys from the beginning and look for the proper span by keeping track of the most recently accessed key for each channel because it is extremely likely that the next access will require either the same key or the very next one.

Once we firstly find the proper span of the channel then we can evaluate the channel's equation at some arbitrary time  $t$ . The main runtime function has to consider four cases for an input time  $t$ , if frame time:

- is before the first key then we use the channel's extrapolation mode.

- is after the last key then we use the channel’s extrapolation mode.
- falls exactly on some frame key then we return its key value.
- falls between two keys then we evaluate span’s cubic equation.

Finally, we provide the file format of our keyframe animation information (Figure 4.6). The file always starts with the keyword “*animation*” following by a single left curly brace “{”. On the next line, we find the keyword “*range*” following by two numbers which describe the start and the finish of the animation time. We further find the keyword ‘numChannels’ on the next line following by the number of the scene’s DOFs. A keyframe information section follows for every channel. The form of this section starts with the keyword ‘*channel*’ following by the DOF identification number and a single left curly brace “{”. On the following line, we find the keyword “*extrapolate*” following by two numbers which specify how the curve will be extrapolated before the first and after the last key frames respectively. Moreover, one line below, we find the keyword “*keys*” following by the number of the key frames and a single left curly brace “{”. The following lines are the descriptions for all possible key frames. We describe each keyframe by defining its time, its value and the incoming and outgoing tangent rules. Finally, the end of any key, channel or animation definition is denoted by a right curly brace “}”.

```

animation {
  range [time_start] [time_end]
  numchannels [num]
  channel [id] {
    extrapolate [extrap_in] [extrap_out]
    keys [numkeys] {
      [time] [value] [tangent_in] [tangent_out]
      .
      .
      .
    }
  }
  channel ...
}

```

Figure 4.6: Keyframing animation file format.

### 4.3 Forward Kinematics Skinning System

In this animation style we use a model to control motion. The animator is responsible for setting up the rules of the model and animation is done by developing motion according to the rules of the model. Complex motion can be attained, but detailed control of the motion is removed from the hands of the animator. Much model-based animation is based on physical principles since these are what give rise to what people recognized as realistic



motion. However, other models can be used. Physically-based modeling is a special case of the more general constraint modeling. Arbitrary constraints can be specified by the user to enforce certain geometric relationships or relationships appropriate for a specific application area. Mental models can also produce animation based on action-reaction scenarios, problem solving and similar “intelligent behavior”.

As we mention earlier, the figure model intended for animation typically consists of at least two main layers. The motion of a highly detailed surface representing the outer shell or skin of the character is what the viewer will eventually see in the final product. The skeleton underneath it is a hierarchical structure of joints which provides a kinematic model of the figure and is used exclusively for animation. Thus, figure global motion (translation and rotation) and *pose* (joint angles) is applied to the skeleton and the skin follows the skeleton motion. Each of the skeleton’s joints acts as a parent for the hierarchy below it. The root represents the whole character and is positioned directly in the world coordinate system. We have proved in previous section that the quaternion outperforming the rotation matrix structure at most all cases, except of the vector transformation occasion and it will be our choice for representing joint’s orientation. So, instead of using a transformation matrix, we can use a quaternion  $Q$  and a translation vector  $T$  ( $QT$  pair). If a local transformation  $QT$  which relates a joint to its parent in the hierarchy is available, one can obtain a transformation which relates local space of any joint to the world system (i.e., the system of the root) by simply concatenating transformations along the path from the root to the joint. To evaluate the whole skeleton (i.e., find position and orientation for all joints), a depth-first traversal of the complete tree of joints is performed. Each joint first computes its new local transformation  $QT L(L_q, L_t)$  based on the previous  $L$  state and the values of its DOFs:

$$\begin{aligned} L(L_q, L_t) * &= T_{joint}(\varphi_1, \varphi_2, \varphi_3), \text{ where } T_{joint}(\varphi_1, \varphi_2, \varphi_3) = (T_q, T_t) \Rightarrow \\ L(L_q, L_t) &= (L_q + L_t)T_q + T_t = L_qT_q + L_tT_q + T_t \Rightarrow \\ L(L_q, L_t) &= (L_qT_q, L_tT_q + T_t) \end{aligned}$$

We can evaluate the transformation  $QT T_{joint}$  by rotating with  $Q_{align}$  quaternion so the *parent axis*, a vector going from joint position ( $J_{pos}$ ) to parent’s kernel centroid aligns with a unit vector  $k$  which lies on positive z-axis, rotate then along the x-, y-, and z- axes through  $\varphi_1, \varphi_2$ , and  $\varphi_3$  respectively (quaternions  $Q(X, \varphi_1)$ ,  $Q(Y, \varphi_2)$ , and  $Q(Z, \varphi_3)$ ) and finally re-position parent axis to the initial location by multiplying with the negative of  $Q_{align}$  quaternion (for more explanation see Quaternions operations at subsection 2.2.2 and Figure 4.7):

$$\begin{aligned}
T_{joint} &= (T_q, T_t) \Rightarrow \\
T_q &= (-Q_{align})Q_{rot}Q_{align} \\
T_t &= (-J_{pos})T_q + J_{pos}
\end{aligned}$$

where  $Q_{rot} = Q(X, \varphi_1)Q(Y, \varphi_2)Q(Z, \varphi_3)$

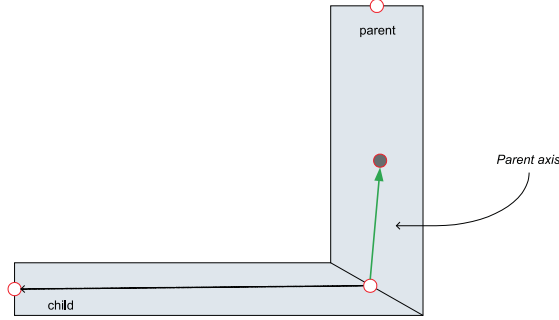


Figure 4.7: Parent axis vector.

Then, the world  $QT W(W_q, W_t)$  is computed for each joint by concatenating its local  $QT L$  with the world  $QT$  of the parent joint: World  $QT W(W_q, W_t) = W_{parent} \cdot L$ . However, since our models consist of a huge number of vectors which transforming in time, it is proper after evaluating the world quaternion to convert it to matrix form and then use the matrix to transform the vectors. Although this general and simple technique for evaluating hierarchies is used throughout computer graphics, in animation it is given the name *Forward Kinematics (FK)*. To animate with this technique, rotational parameters for all joints involved in the motion must be set explicitly by the user, using the pre-specified scripting animation language (Figure 4.8).

We can see that no account is taken of any physical laws that might be in effect on the figure. So, it will be proper to include natural constraints for real looking animation results. What makes it possible for things to be built in the real world is that fact that things are solid and objects can't pass through one another. Incorporating collision detection is one of the more realistic aspects of physically based modeling. However, time in computer graphics is discrete and most collision detection takes place at discrete intervals. This means that it is possible to miss a collision altogether and certainly probable that it won't be detected at the instant of contact. In this work, a motion of a joint is acceptable if the new joint angle is inside the joint's angle range given by the user at our BVH file and if all its sub-tree components do not collide with any other component. If any of two comes true then the motion is cancelled. A costly but efficient way of specifying if a collision occurs between two components is checking if their BBs collide (see Bounding Box at subsection 2.3.1).

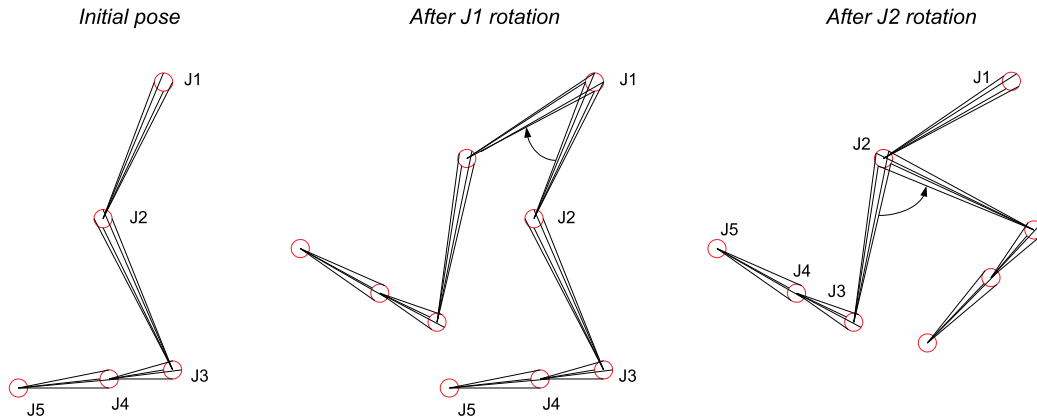


Figure 4.8: Forward kinematics example.

However, the number of the BB collision function calls equals to the product of the number of the sub-tree nodes and the remaining nodes, making this solution very slow. Since this number is huge, we propose to make a first collision approximation by finding which components probably collide using bounding spheres (see Bounding Sphere at subsection 2.3.1) and then check if there is actually a collision between each pair using the BB collision function. Although, we have reduced the possible collision component pairs a lot, we can speed our method by saving a weight at any pair which specifies the collision possibility. We evaluate this weight as  $w = (l/r)^2$  where  $l$  is the distance between their sphere centers and  $r$  is the sum of their radius (Figure 4.9). So, if we order these pairs by their weights then we will find or not a collision with the less tries. Finally, it's smart and will speed up our method if, for each component, have pre-computed the components which are not at its sub-tree.

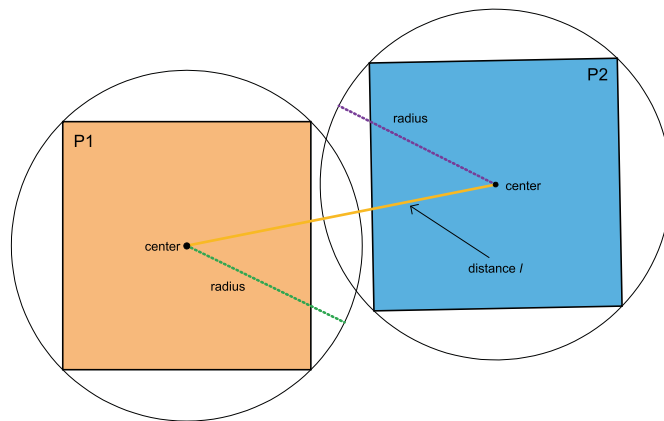


Figure 4.9: Bounding sphere collision example.

Every successful motion is transferred to the figure's surface by assigning each skin vertex one joint as driver (*rigid skinning*). A skin vertex is simply frozen into the local space of the corresponding joint, which is the one chosen directly by the skeletonization procedure.

The vertex then repeats whatever motion this joint experiences, and its position in world coordinated is determined by standard FK procedure, namely is transformed by the joint's world matrix:  $v' = Wv$ . Similarly, the normal of each vertex is transformed by only rotating with the world matrix:  $n' = Rn$ ,  $W = RT$ . Although it is simple, rigid skinning makes it difficult to obtain sufficiently smooth skin deformation in areas near the joints (Figure 1.2) or also for more subtle effects resembling breathing or muscle action. At the next section, we will propose a method for solving the collapsing problem in areas near the joints.

In this section we discussed that the geometric defects of vertex animation can be overcome by using skeleton animation. Skeleton animation, as the name implies means using an articulated structure as the framework that admits the animation control; then attaching a mesh to the skeleton which provides the material to be rendered. We further include natural constraints to our system for making more real looking animation results using collision detection and joint limitation subsystems. Finally, Figure 4.10 shows a diagram of our forward kinematics skinning system.

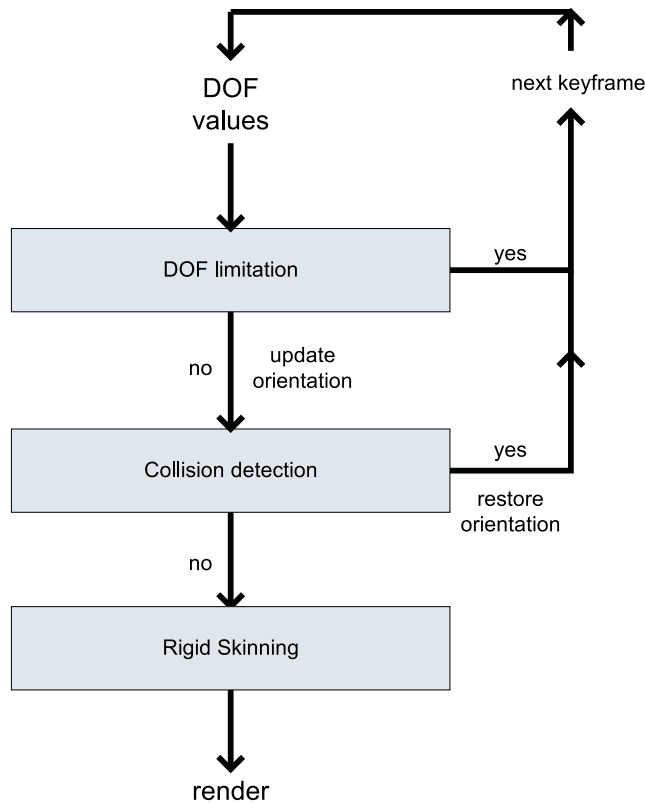


Figure 4.10: The forward kinematics skinning framework.

## 4.4 Gap Reconstruction Process

The basic skinning approach has many flaws and shortcomings. The major one is the difficulty to obtain sufficiently smooth skin deformation in areas around the joints. When a motion of a component occurs, a part of this component collides with its parent resulting in triangles of each component to be inside each other. Also, one may notice that a gap is created at the other side of the collision location. This serious discontinuity between segments results in unacceptable skin deformations. This has led to the emergence of more specialized techniques that improve the final shape result. We have developed a technique to obtain sufficiently smooth skin in areas near the joints after the influence of the skinning procedure. This technique can be divided into four sub-problems:

1. Remove skin vertices of overlapping components.
2. Add new vertices in this area to fill the gap.
3. Construct a blending mesh that produce a derive smooth skin mesh by using a robust triangulation method.
4. Compute and adjust surface normal vectors.

In the experimental evaluation, the results are promising and the proposed technique exhibits efficient and robust behavior. Our main concern of this method is to limit the high computation time of the skin construction in the area near the joints which is due to the triangulation level. Algorithm 5 summarizes the entire process.

#### 4.4.1 Remove Vertices

The first step of this technique is to detect which points from each of the collided components are located inside the other one. The most accurate method is to check whether each point of one component is inside the other one or its convex hull. However, since both component and its convex hull consist of a large number of triangles, this process may require a lot of CPU resources. The oriented bounding box collision test is not very precise on complicated objects, since it has a box-shape, but despite of all this it is extremely fast, and easy to use. A point is inside an oriented bounding box if all 6 plane-point tests results have the same sign. Since that the OBB collision test is not so accurate, we know that a large set of points that is outside the component will be not sent for rendering.

However, we can improve our result by testing each point with one extra plane (Figure 4.11). It's the plane which fits best to the common (opening) points of these components and can be computed by the principal component analysis method discussed in Appendix C. We can calculate the covariance matrix for the set of points and then determine the eigenvectors corresponding to the two largest eigenvalues. These two vectors are parallel to the best-fit plane. Then, take either their cross product or the eigenvector

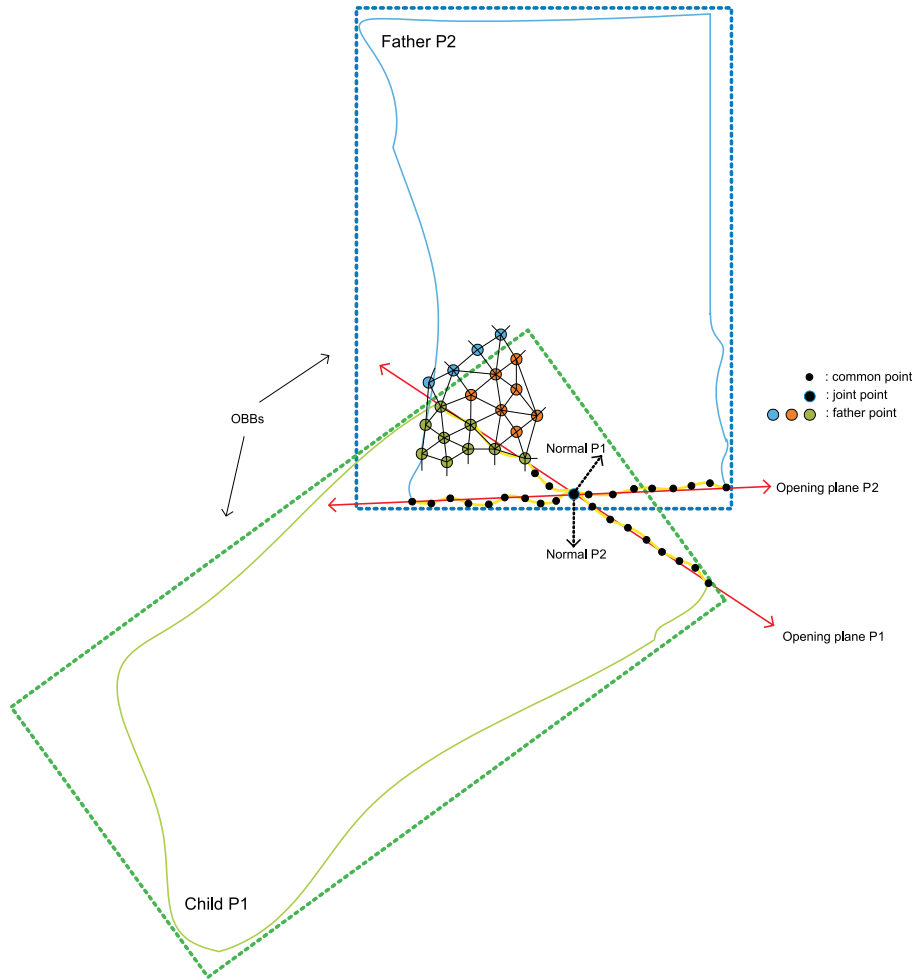


Figure 4.11: This is an example which shows the difference of testing with and without opening plane (red lines). If we are testing the parent points with the child’s OBB without the opening plane, then it will result to remove the green and orange points. On the other hand, the cut is more accurate since the removed points are only the green ones.

corresponding to smallest eigenvalue to evaluate the plane’s normal. Finally, we adjust the plane so that it passes through the centroid of point set.

From all component points only a small percentage of them have probability to be removed. So, we propose to start from the opening points which have the highest possibility to be inside to the other component’s oriented bounding box and move through their 1-ring of neighbors until we find the points which are outside the OBB (Figure 4.12). Point’s 1-ring of neighbors are the points which belong to the same facet. We will speed up our method if, for each point, we have pre-computed its 1-ring of neighbors. So, this method provides the perfect cut of the component executing the fewer tests. Moreover, we store the outside (end) points because they are necessary for calculating the patch which will fill the gap. As we will discuss below, there are two categories of end points which have different patch construction methods, the points that belong to the component’s opening (*Circle points*) and the others that have been found through the iterative method

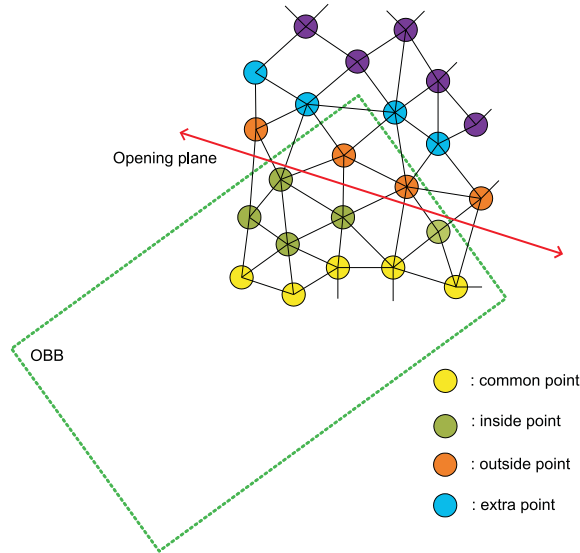


Figure 4.12: This is an example which shows the point removing process. We start from the opening points (yellow) and continue with their adjacents until we find an outside point. When we have found all end points (orange) then we maintain their neighbor information (purple) for use in the triangulation method.

(*Bezier points*). The created patch many have discontinuities (holes generation) in one or both components because the triangulation process did not use one or more of the given end points. Therefore, we must extend the triangulation by adding the neighbors of the end points (*Extra points*) to ensure the robustness of the constructed patch. Finally, we group the component's opening points which have been removed (*Removed points*) (Figure 4.13).

#### 4.4.2 Add New Vertices

Once we have classified the end points from both components, we should patch the generated holes. The objective is to estimate the points which provide sufficiently smooth skin deformation in this area. To do so, carry out this process into two steps:

- estimate new points by using the child's Circle points (denoted by  $S_1$ )
- estimate new points by using both child's and parent's Bezier points (denoted by  $S_2$ )

The observation behind  $S_1$  is that when we rotate a component about an arbitrary axis, movement of every *Circle* point  $v$  will describe a circular arc (Figure 4.14). The task is to find new points by interpolating from the point's position before the rotation  $q_0$  ( $v_0 = q_0 v q_0^{-1}$ ) to the point's position after rotation  $q_1$  ( $v'_0 = q_1 v q_1^{-1}$ ) (see Figure 4.14). We

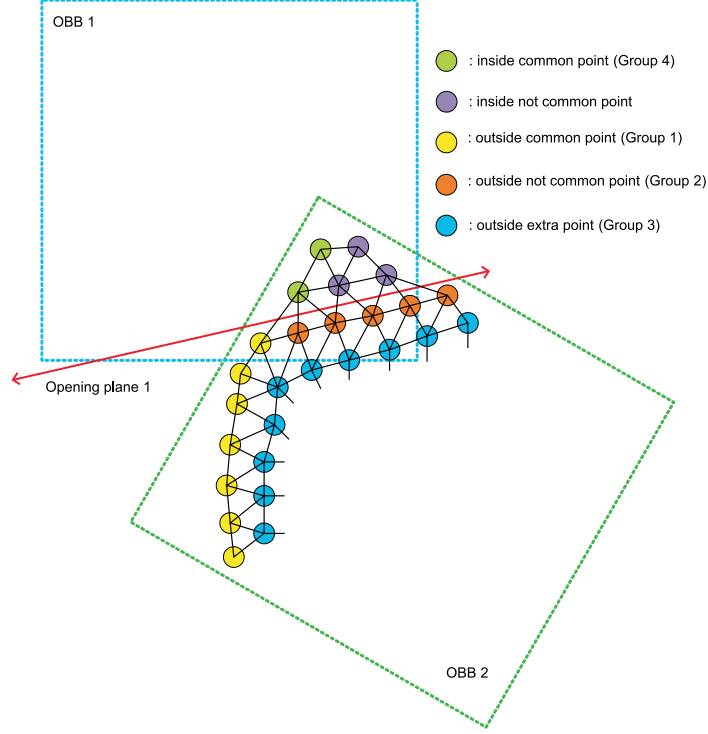


Figure 4.13: This is an example which shows the point classification. Group 1 is the Circle end point set, Group 2 is the Bezier end point set, Group 3 is the Extra point set, and finally the Group 4 is the Removed point set.

use fast linear interpolation of quaternions (QLERP) since the interpolation along the shortest segment hardly causes any observable skin defect [51]. QLERP is computed as

$$q(t; q_0, q_1) = \frac{q'}{\|q'\|}, q' = (1 - t)q_0 + tq_1 \quad (4.1)$$

where  $t$  is the interpolation step and depends on the proper distance which must be assigned to constructed points. We evaluate it as the average of the median edge distances of the two components. To avoid quaternion's normalization  $sqrt$  operation, we convert  $q'$  to a homogeneous rotation matrix  $Q'$  and then normalize subsequently by dividing with  $q'q'$  to provide  $Q$ . In order to change the center of rotation from the joint position ( $j_{pos}$ ) to origin, we define a homogeneous matrix

$$T = \begin{bmatrix} I & -j_{pos}^{\vec{}} \\ \vec{0}^T & 1 \end{bmatrix} \quad (4.2)$$

where  $I$  is a  $3 \times 3$  identity matrix. Finally, a new point can be constructed using the following formula:

$$V' = TQv_0T^{-1} \quad (4.3)$$

We have to fill in the gap in the area where we have removed points from both components. The major issue of  $S_2$  is that we only have the *Bezier* points from each component without any correspondence between them. So, we propose to use as extra point set, the *Middle*



points. *Middle* is the point set which consists of the average of the initial and final positions of the *Removed* points.

A grouping function is developed to create *triplets* from each *Middle* point and one point from child's and father's *Bezier* sets. The feature of each triplet is that the plane defined by its points, has the minimum dihedral angle with the plane defined by the joint position and the kernel centroids of the participating components (plane *PL*). If we use to fill the gap with points created from circle equation using the triplets, artifacts will arise due to the discontinuities at the end points of the two components. To avoid these shortcomings, we construct blending arcs given the triplet as control points using a Rational Bezier representation (see subsection 2.1.3).

The tangent vectors at *Bezier points* defined as the projections on *PL* of the vectors going from the joint position to child's (*child axis*) and parent's (*parent axis*) kernel centroids, respectively. To prevent the generated meshes from exhibiting volume loss as joints rotate to extreme angles we replace *Middle* point. We propose to use the average of the *Middle* point and the intersection of their tangents and then evaluate the tangents to the vectors going from the *Bezier points* to the *Middle* point (see Figure 4.14). Finally, we ensure that the constructed points are uniformly sampled by adjusting the interpolation step.

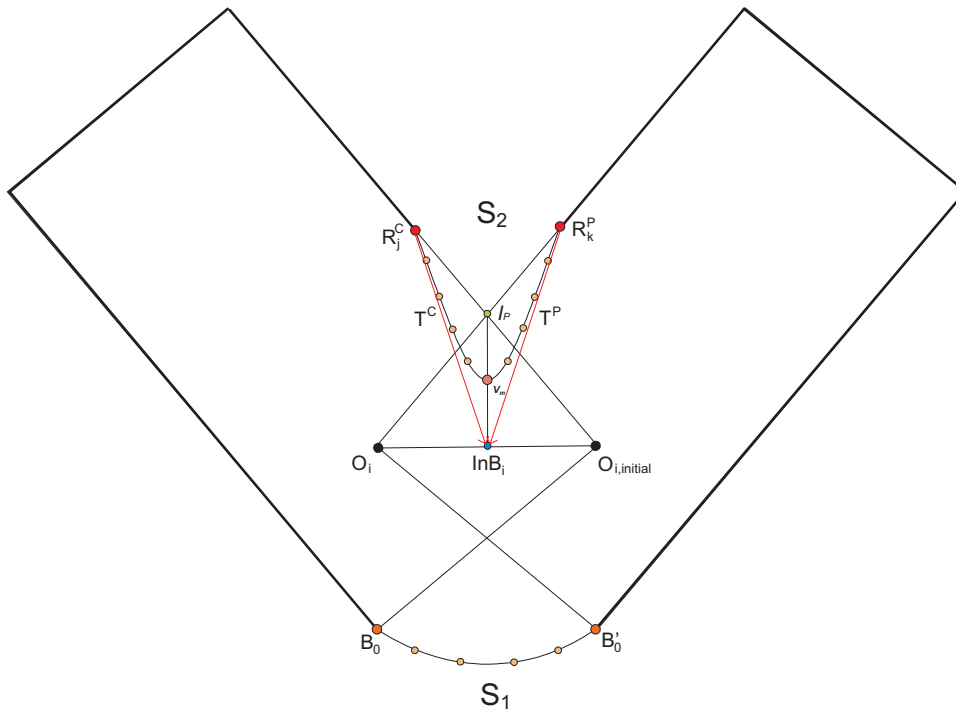


Figure 4.14: Introducing additional points and blending curves.

### 4.4.3 Triangulate Vertices

The points derived through the process of the above routines constitute the scattered 3D data which will be triangulated. To ensure the robustness of the constructed patch we extend this set by adding the 1-ring neighbors of the *Bezier* points of the components.

In this work, we use the *Tight Cocone* algorithm [28] that constructs water-tight surfaces. The algorithm works on an initial mesh generated by the popular surface reconstruction algorithm *Cocone* [5] and fills up all holes to output a water-tight surface. In doing so, it does not introduce any extra points and produces a triangulated surface interpolating the input sample points. Since we need only two parts of the generated spheroid to fill the holes, we must eliminate a number of unnecessary facets to derive optimal lighting results. First, we approximately correct the normal vectors orientation and then we remove the facets that belong exclusive to either the child or the parent component and whose normal vector is not nearly parallel with all normal vectors of its defining points (Figure 4.15).

#### 4.4.4 Compute Normal Vectors

Finally, we need to calculate the normal vectors of the output surface and correct the normal values of the components end points (Figure 4.15). The evaluation of the constructed point normals is achieved by averaging the normals to the output surface facets that contain the point. The normal vectors of the components end points are calculated by averaging the normals of the output surface facets that share the point and the normals of the preexistent component facets that contain the point. However, we should be cautious not to exclude component facets that have been removed during the second process. Finally, the normal vectors of the Extra point set remain unaffected since the above adjustment is not necessary.

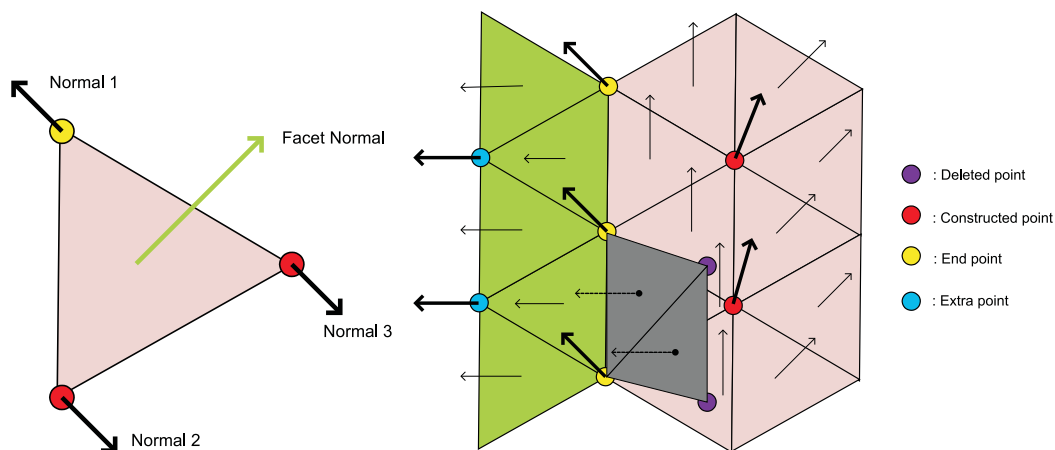


Figure 4.15: (Left) A rejected facet. (Right) The normal evaluation of all possible point cases. Component facets are shown in green color, deleted facets are shown in grey and constructed facets are shown in pink color.

---

**Algorithm 5** advanced\_rigid\_skinning()
 

---

1.  $V = B^C = B^P = R^C = R^P = \{\emptyset\}$ ;
2.  $Q^C = \{OP_1^C, \dots, OP_{n_C}^C\}$ ; remove( $V, Q^C, B^C, R^C$ );
3.  $Q^P = \{OP_1^P, \dots, OP_{n_P}^P\}$ ; remove( $V, Q^P, B^P, R^P$ );
4. **for each**  $B_i^C \in B^C$  **do**
5.      $V \cup = \{v'_t(B_i^C, q_0, q_1, T) : \text{eq}(4.1), \text{eq}(4.3)\}$ ;
6. **for each**  $O_i \in OP^C - B^C$  **do**  $InB_i = \frac{O_{i,initial} + O_i}{2}$ ;
7. **for each**  $InB_i \in InB$  **do**
8.      $(R_j^C, R_k^P) = \text{triplet}(InB_i, R^C, R^P, PL.N)$ ;
9.      $I_P = \text{intersection}(P_{PL}(axis^C), P_{PL}(axis^P))$ ;
10.      $V_m = (I_P + InB_i)/2$ ;
11.      $T^C = InB_i - R_j^C$ ;  $T^P = InB_i - R_k^P$ ;
12.      $V \cup = \{c_t(InB_i, R_j^C, R_k^P, T^C, T^P) : \text{eq}(??)\}$ ;
13.  $B = B^C \cup B^P$ ;  $R = R^C \cup R^P$ ;  $V \cup = NR_1^*(R)$ ;
14.  $F = \text{triangulate}(V)$ ;
15. **for each**  $F_i \in F$  **do** correct( $F_i.N$ );
16.  $F - \{F_i \mid \forall 0 \leq k \leq 2 ((V_k(F_i).N \stackrel{\#}{\approx} F_i.N) \text{ and } V_k(F_i) \in (B^C \cap R^C) \text{ or } (B^P \cap R^P))\}$ ;
17. **for each**  $V_i \in V$  **do**
18.     
$$V_i.N = \begin{cases} \frac{\sum_k^n V NR_1^V(V_i).N_k}{\sum_k^n V NR_1^V(V_i).N_k + \sum_k^n S NR_1^S(V_i).N_k} & | V_i \notin B \cap R \\ \frac{\sum_k^n S NR_1^S(V_i).N_k}{\sum_k^n V NR_1^V(V_i).N_k + \sum_k^n S NR_1^S(V_i).N_k} & | V_i \in B \cap R \end{cases}$$
19.     normalize( $V_i.N$ );

**Function** remove( $V, Q, B, R$ )

1. **while** ( $q = \text{Pop}(Q) \neq NULL$ ) **do**
2.     **if** CollisionTest( $q$ ) **then**  $Q \cup = NR_1^*(q)$ ;
3.     **else**  $V \cup = q$ ;
4.     **if**  $q \in Q$  **then**  $B \cup = q$ ; **else**  $R \cup = q$ ;

**Function** triplet( $InB_i, R^C, R^P, PL.N$ )

1. **for each**  $r_i^C \in R^C$  **do**
  2.     **for each**  $r_j^P \in R^P$  **do**
  3.          $A_{ij} = \angle(\Delta(InB_i, r_i^C, r_j^P).N, PL.N)$ ;
  4.  $(R_i^C, R_j^P) = \{(r_i^C, r_j^P) \mid A_{ij} < \{A_{kl}\}_{1 \leq k, l \leq n_{RC}, n_{RP}}\}$
- 

In algorithm 5, we use  $C$  and  $P$  for child and parent components. Let  $B^C, R^C$  and  $InB$  be Circle( $C$ ), Bezier( $C$ ) and Middle sets. Moreover,  $Q^C$  is a queue containing child's opening points ( $OP^C$ ).  $P_{PL}(axis^C)$  is the projection of *chils's axis* on plane  $PL$ . We define  $NR_1^*(X)$  and  $NR_1^V(X)$  as the 1-ring neighbors of  $X$  set on components and blending mesh, respectively. Let cardinalities of  $NR_1^*(X)$  and  $NR_1^V(X)$  be  $n_*$  and  $n_V$ . In addition, the normal vector of a point or a facet is denoted by “ $N$ ”.  $\Delta(V_a, V_b, V_c)$  is the triangle defined by  $V_a, V_b$  and  $V_c$  points and  $V_a \stackrel{\#}{\approx} V_b$  symbolize when vectors  $V_a$  and  $V_b$  are close to parallel. We also define the same for the parent component. Finally, we denote the patch points and facets as  $V$  and  $F$ .

# CHAPTER 5

## IMPLEMENTATION AND RESULTS

---

### 5.1 Implementation Details

### 5.2 Experiment Results

---

In this section we discuss the implementation details of the system we have developed that performs skeletonization and animation of three-dimensional solid modular models. We will demonstrate the feasibility of the effort and the efficiency and robustness of our approach.

### 5.1 Implementation Details

The programming language and the supporting libraries for the develop of the system was key issue worthy of careful consideration. The main requirements were fast object code production, an effective graphical user interface development tool to enable us to interact efficiently with the user and last but not least support for accurate mathematical computations.

Our language choice for this particular project was OpenGL API tied to C++. Note that for GUI development and scientific libraries, there are usually several available choices. There is a variety of available GUI toolkits for OpenGL API such as *Qt*, *OpenGL Utility ToolKit (GLUT)* [105], *Fast Light ToolKit (FLTK)* and others. We selected the GLUT graphical interface toolkit since it can support efficient user interaction. Similarly, the availability of scientific libraries for C++ is good but much cluttered. There are several libraries that do different or the same things, some better than the others, and it is quite hard to find the best solution. WE decide to use *LAPACK* library [106] as it provides sufficient support for the majority of linear algebra problems that we faced in this thesis. Moreover, through the development of our system we employed additional stand-alone

software, the *qHull* library [108] and the *Tight cocone* [28] program for computing the convex hull and the triangulation of a set of points as we discussed in subsection 2.3.2 and section 4.4.3. Finally, this software form allows us to make the application portable across platforms. This means that the application can run in a variety of platforms, from Windows to many UNIX clones with X-Windows based graphical environments.

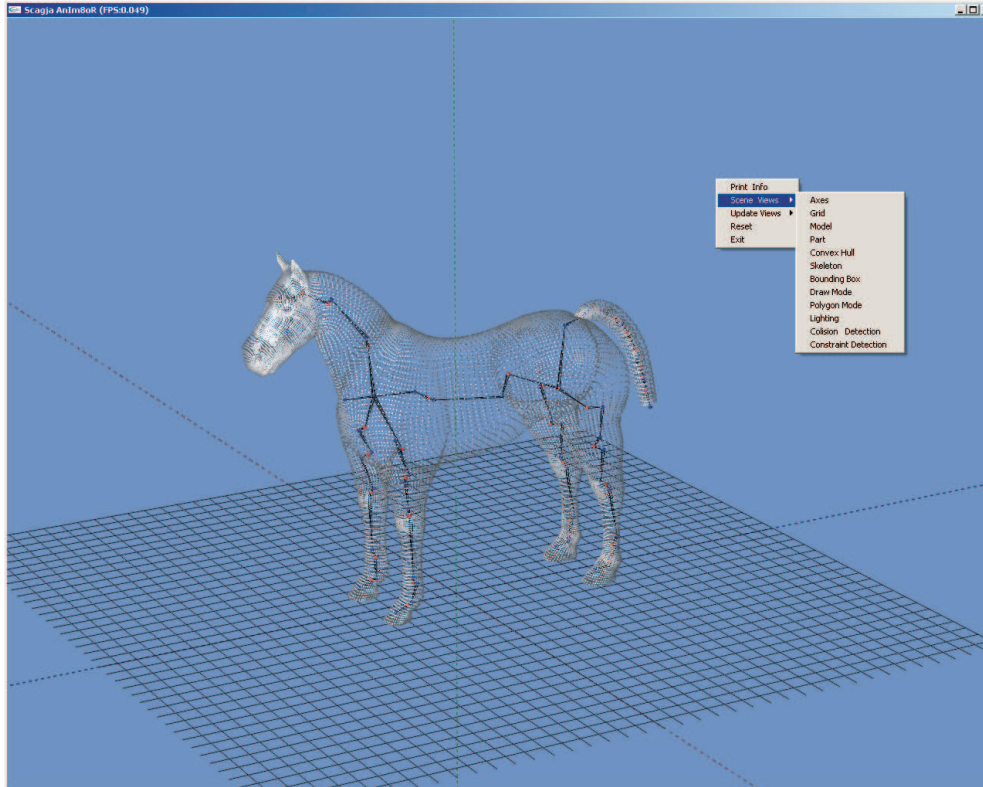


Figure 5.1: The main graphical user interface: We see a horse model rendered by its vertices that facilitate the view of its skeleton representation.

In Figure 5.1 we can see how the main graphical user interface looks while running on Windows XP. As we can see the GUI is very easy to use and aim to visualize the extracted skeletal results and the produced animated models. The user can load and work on one model at a time, but may use a set of animation moves by resetting the model to the initial position. This project requires a *3D Studio Max* [99] file format model, its associated *BVH* file format with the structure discussed at section 3.1, a set of available animation moves (*CLIP* file format) and the skeletonization method. We have three skeleton extraction methods: opening, centroid and principal axis. The necessary inputs for the program execution are the model, the components hierarchy tree and the parameter that defines the skeletonization method. The animation file can be loaded at a later time after the model has been loaded. When the main graphical user interface of the application has opened, we can see a colorful coordinate system (x-axis: red, y-axis: green, z-axis: blue) and a grid floor where the model is placed on. Our program provides users with a number of actions to select from by using the keyboard and the mouse. Some of these interaction

actions are:

- Move or rotate the camera position.
- Print model’s geometry information on command window.
- Select a model’s component (which results to see its OBB).
- Change the model’s geometry rendering (which results to see the skeleton representation).
- Load the animation file.
- Reset model’s components positions to their initial positions (if any animation has came true).
- Exit from the program.

## 5.2 Experiment Results







In this section we present the experiment results of our work. The experiments are used to demonstrate the performance of the proposed methods. All experiments are performed on an Intel Core Quad 2.4 GHz CPU where only one core was in use with 2048 Mb RAM using an ATI Radeon 3870 graphics card. A summary of the studied models, which includes several articulated game characters and animals [109], is shown in Table 5.1. Size is measured as the number of vertices and the triangles of each model.

First, we will present and analyze the extracted skeletal results using the three aforementioned methods. Then, the extracted skeletons can be used for animation. Finally, we present sequences of images obtained from our principal-axis skeleton-based animation approach of several objects using hand-made motion data.

### 5.2.1 Skeletonization results

As we discussed above, these experiments are used to demonstrate the performance of the proposed skeletonization methods. We see that the executive time of the Opening method depends only on the number of their components, which practically means constant complexity. So, Opening method is independent from the vertices and triangles cardinalities, making it the same fast for any model. Moreover, it appears that the processing time of the other two skeletonization methods depends on the number and the size of their components. Centroid method takes  $\sum_{i=0}^k O(n_i \log n_i) + O(r_i \log r_i)$  due to the kernel’s centroids computation (where  $k$  is number of components,  $n_i$  and  $r_i$  are vertices and kernel vertices respectively of  $i$  component), which makes it more expensive to find than the first method.

Table 5.1: Experimental models.

Model	Parts	Vertices	Triangles	Figure
Cow	16	3825	7646	
Homer	9	4930	9856	
Dilo	16	5524	11044	
Camel	27	6013	12022	
Horse	17	8964	17924	
Dino	21	9731	19458	

Furthermore, we derive skeletons using our Principal Axis method in  $\sum_{i=0}^k O(n_i \log n_i)$  time due to principal axis estimation (where  $k$  is number of components and  $n_i$  are vertices of  $i$  component) plus the kernel’s centroid computation time which we estimated above. We observe from measures that skeleton extraction complexity appear to be nearly linear on the number of triangles. Examples show that the expensive part of this method is the kernel’s centroid computation since the principal axis estimation has high performance. Moreover, the time cost of grouping opening centroids using four or less extra points and executing the proposed local and parent refinement methods is negligible compared to the overall performance (Table 5.2).

Our method generates refined skeletons in less than half a minute for dense models. Thus, if our method is used in conjunction with a fast decomposition method, it will result in a very efficient overall process. Finally, Table 5.2 summarize the cumulative time performance (measured in seconds) of the methods discussed. We observe that the first method is the fastest one, the second method is more expensive (due to kernel computation), and finally, the last technique costs slightly more than the not refined techniques.

We provide the skeletonization results using the three alternative methods for every model. Figures (5.2, 5.5, 5.8, 5.11, 5.14, 5.17) illustrates the results of the Opening and Centroid methods. Moreover, skeletons extracted with Principal Axis method are illustrated in figures (5.3, 5.6, 5.9, 5.12, 5.15, 5.18) using only the principal axis (Left) and Local Refinement (Right) and in figures (5.4, 5.7, 5.10, 5.13, 5.16, 5.19) using all

principal directions (Left) and Parent refinement (Right). Although there are no well criteria to measure the skeletal quality quantitatively, it is obvious from all produced skeletons the superiority of our Principal axis method over the other methods.

Table 5.2: Time performance of skeletonization methods. Columns from left to right represent time performance using Opening method, Centroid method, Principal Axis method using only the covariance-computed PA and our Principal Axis method.

Model	Opening	Centroid	Principal Axis - Only PA	Principal Axis - Our Method
Cow	0.00223	4.86503	4.92670	5.08260
Homer	0.00106	4.40853	4.30518	4.55921
Dilo	0.00164	5.64114	5.77643	5.91370
Camel	0.00342	8.63889	8.66312	8.93177
Horse	0.00193	7.94017	8.21302	8.44414
Dino	0.00256	9.21676	9.27403	9.92523



### Cow Model

- Opening - Centroid Methods

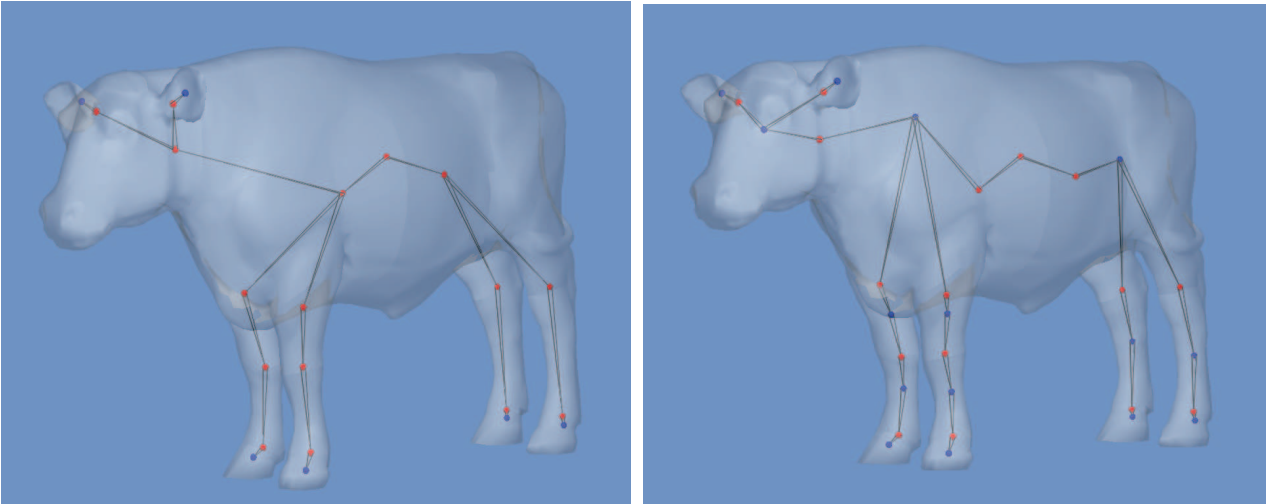


Figure 5.2: Cow model skeleton representation using (left) Opening and (right) Centroid methods.

- Principal Axis Method

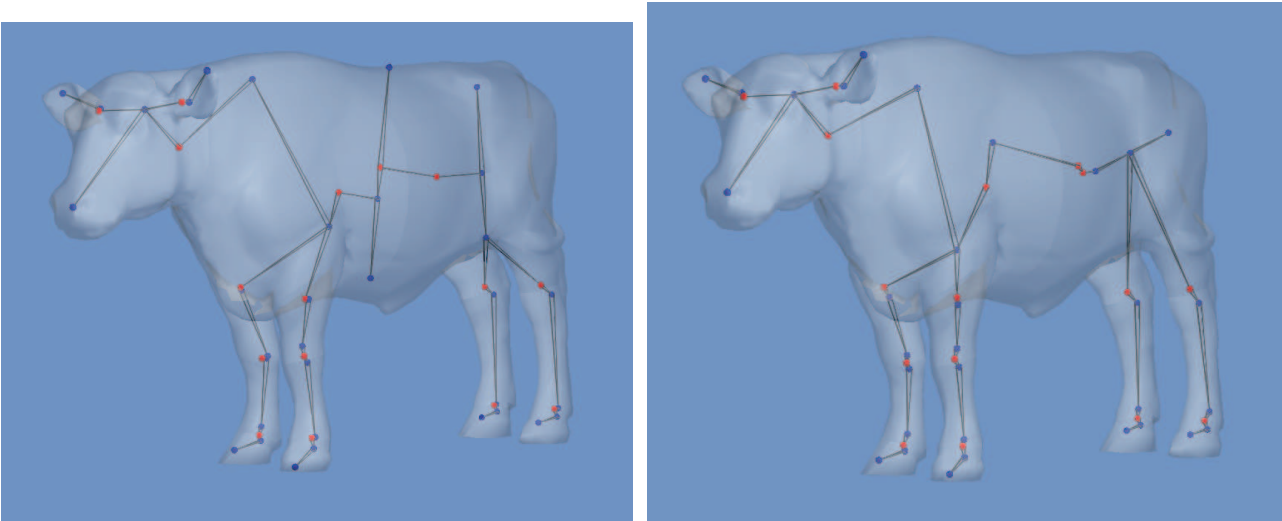


Figure 5.3: Cow model skeleton representation using only principal axis (left) without and (right) with Local Refinement (12 degrees)

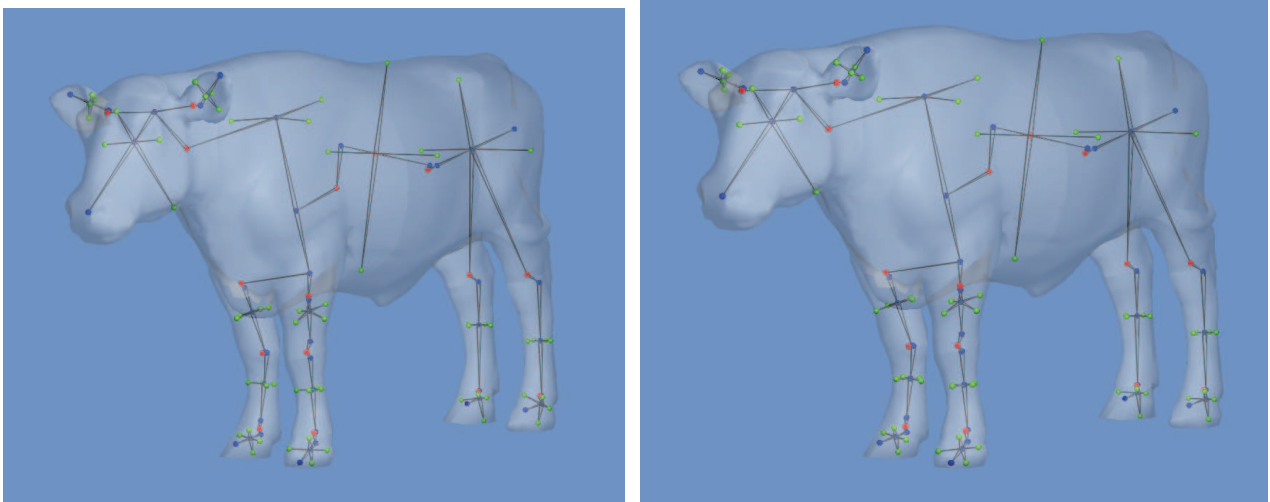


Figure 5.4: Cow model skeleton representation using all principal directions (left) without and (right) with Parent Refinement

## Homer Model

- Opening - Centroid Methods

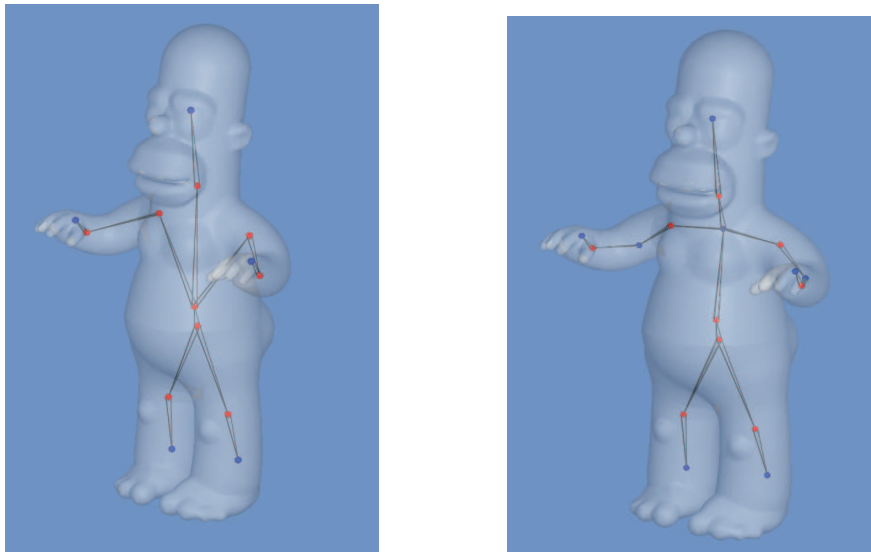


Figure 5.5: Homer model skeleton representation using (left) Opening and (right) Centroid methods.

- Principal Axis Method

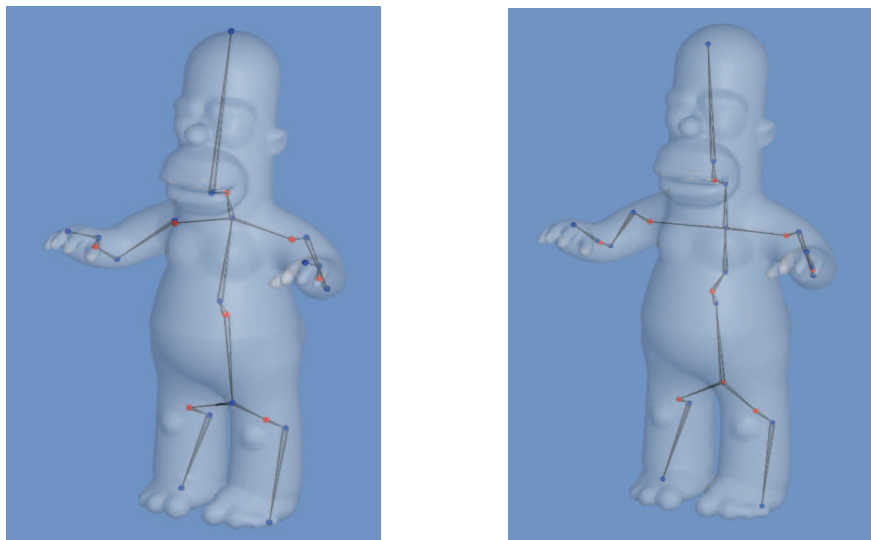


Figure 5.6: Homer model skeleton representation using only principal axis (left) without and (right) with Local Refinement (18 degrees)

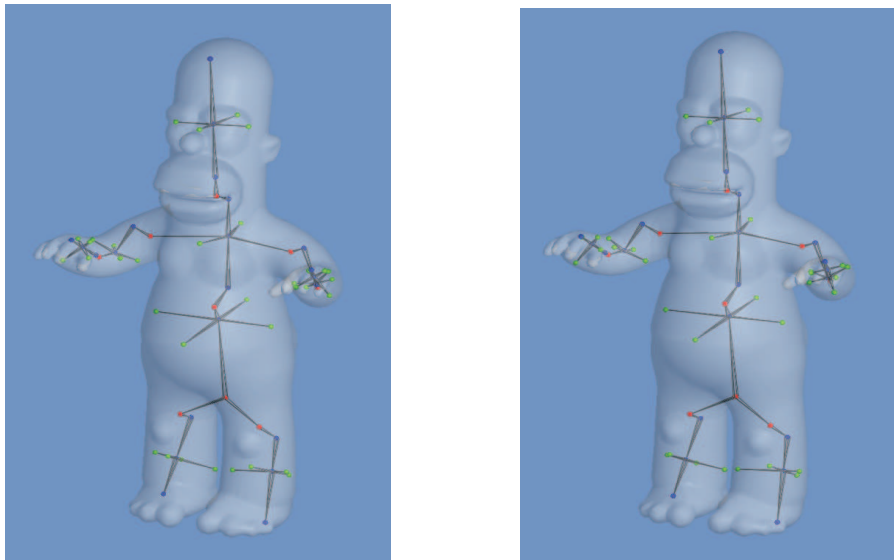


Figure 5.7: Homer model skeleton representation using all principal directions (left) without and (right) with Parent Refinement

## Dilo Model

- Opening - Centroid Methods

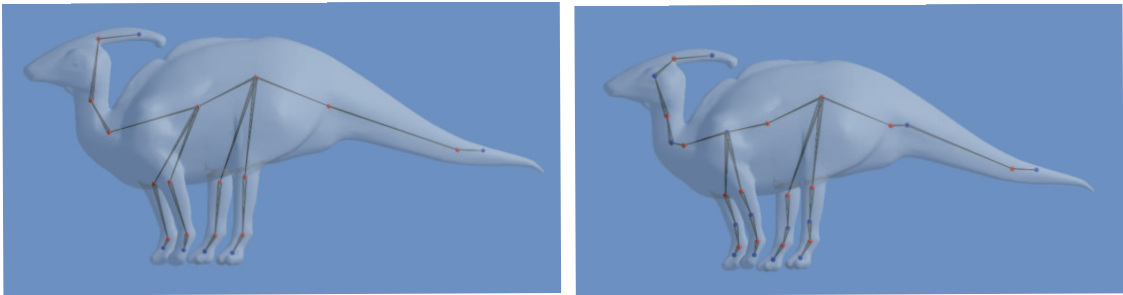


Figure 5.8: Dilo model skeleton representation using (up) Opening and (down) Centroid methods.

- Principal Axis Method

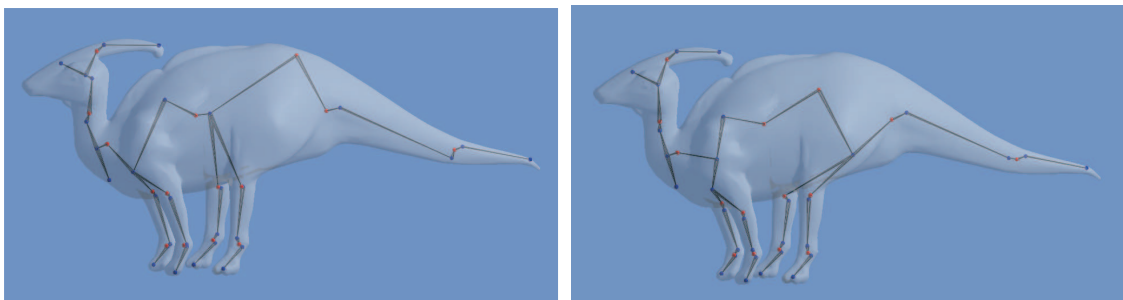


Figure 5.9: Dilo model skeleton representation using only principal axis (up) without and (down) with Local Refinement (10 degrees)

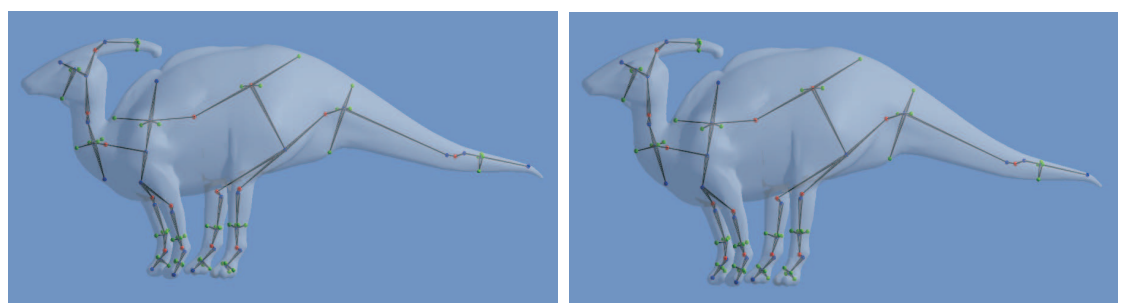


Figure 5.10: Dilo model skeleton representation using all principal directions (left) without and (right) with Parent Refinement

## Camel Model

- Opening - Centroid Methods

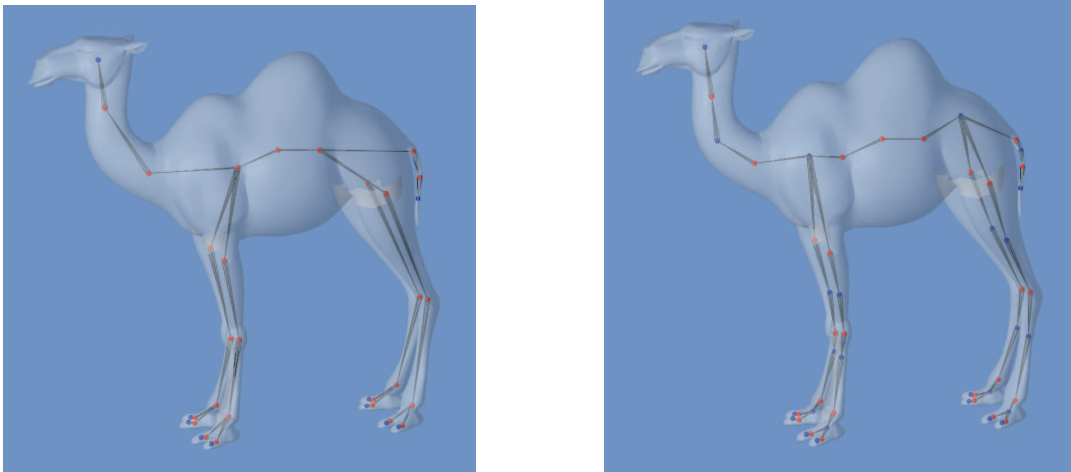


Figure 5.11: Camel model skeleton representation using (left) Opening and (right) Centroid methods.

- Principal Axis Method

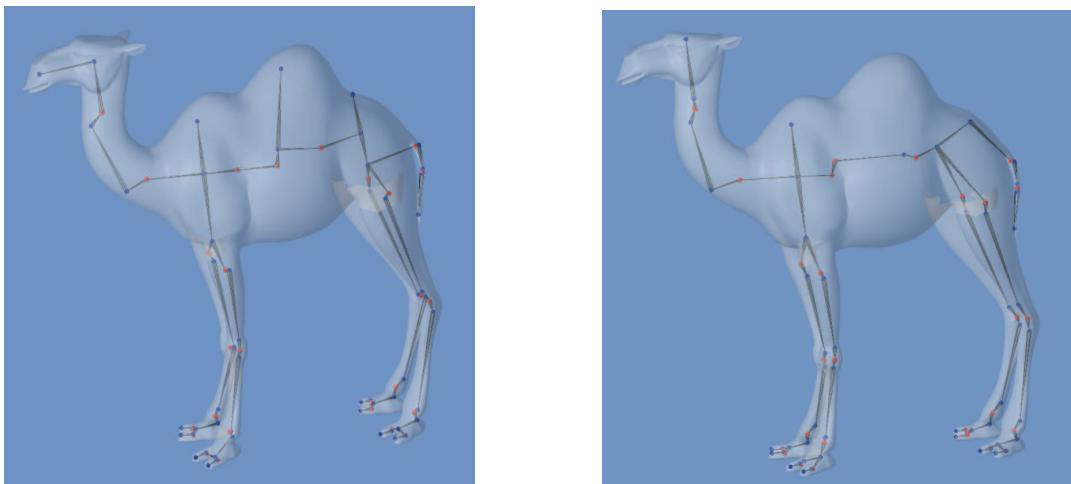


Figure 5.12: Camel model skeleton representation using only principal axis (left) without and (right) with Local Refinement (22 degrees)

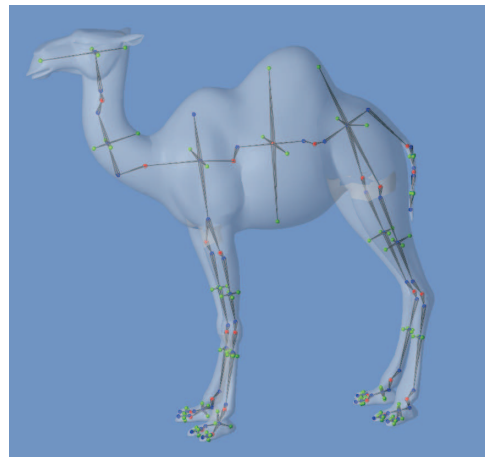
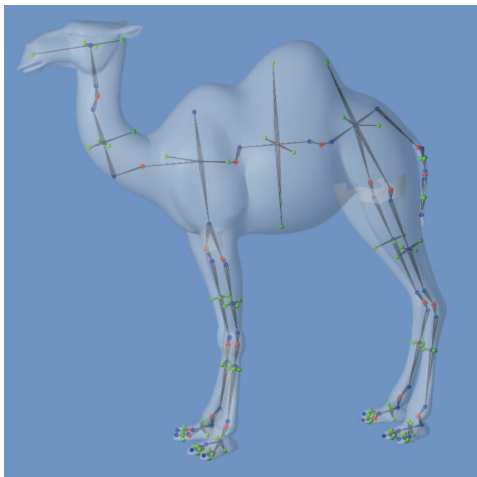


Figure 5.13: Camel model skeleton representation using all principal directions (left) without and (right) with Parent Refinement

## Horse Model

- Opening - Centroid Methods

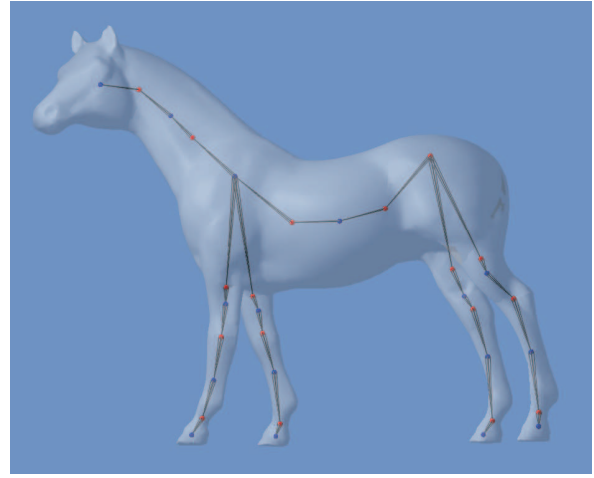
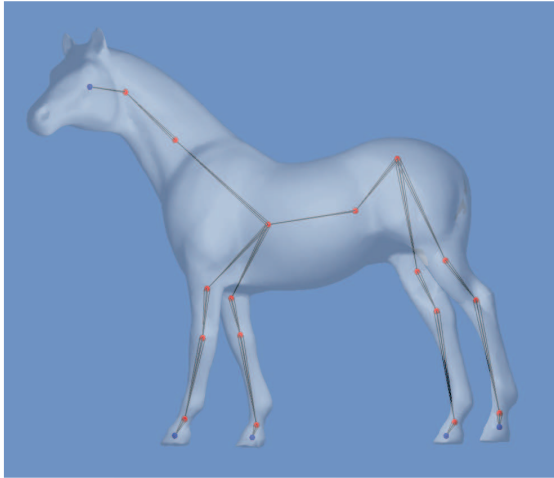


Figure 5.14: Horse model skeleton representation using (up) Opening and (down) Centroid methods.

- Principal Axis Method

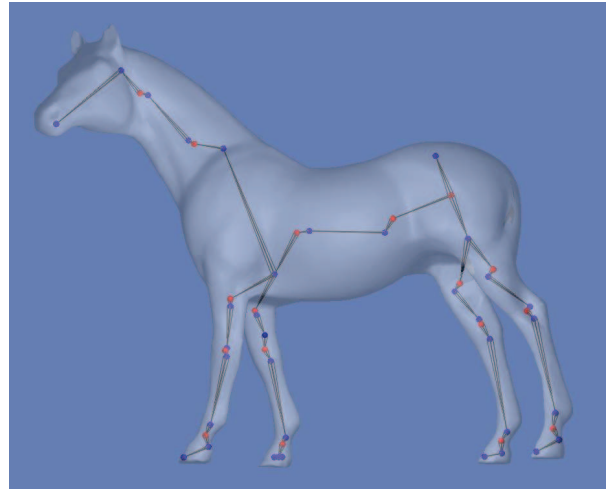
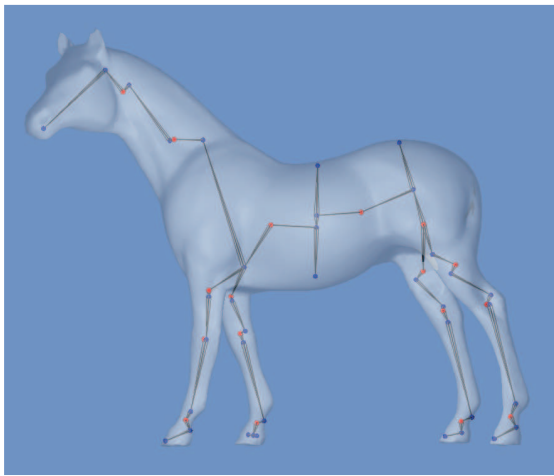


Figure 5.15: Horse model skeleton representation using only principal axis (up) without and (down) with Local Refinement (20 degrees)



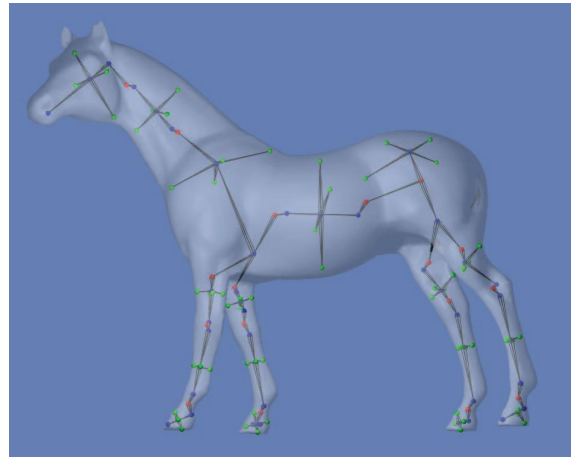
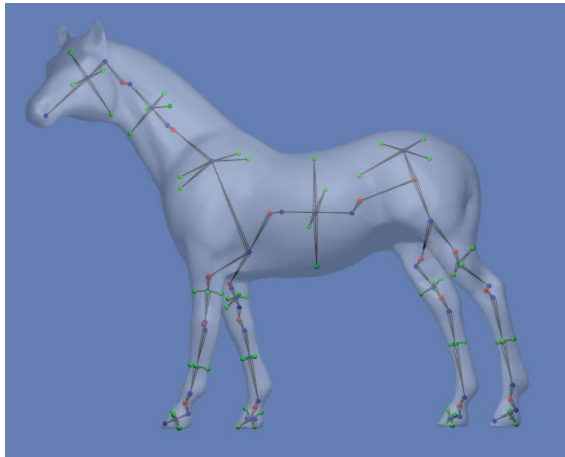


Figure 5.16: Horse model skeleton representation using all principal directions (left) without and (right) with Parent Refinement

## Dino Model

- Opening - Centroid Methods

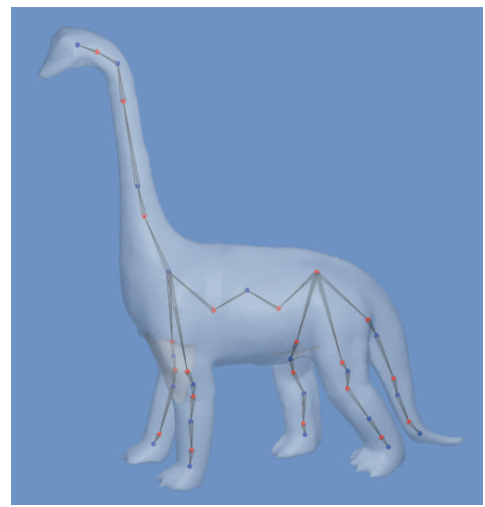


Figure 5.17: Dino model skeleton representation using (left) Opening and (right) Centroid methods.

- Principal Axis Method



Figure 5.18: Dino model skeleton representation using only principal axis (left) without and (right) with Local Refinement (18 degrees)

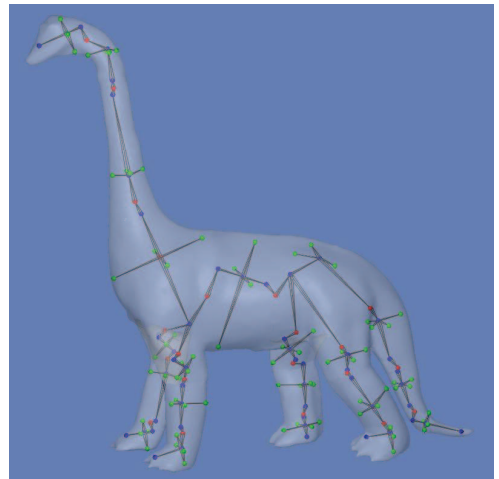
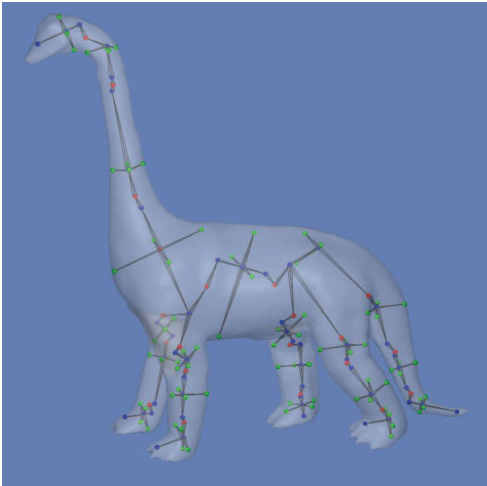


Figure 5.19: Dino model skeleton representation using all principal directions (left) without and (right) with Parent Refinement

### 5.2.2 Animation results

The extracted skeletons can be readily used to create animations. We demonstrate this advantage by re-targeting hand-made motion data to the skeletons extracted from the original mesh using our Principal axis method. The motions, i.e., joint angles, are used as motion data of the skeletal joints. Incoming and outgoing tangent rules are set to *Flat* for creating “slow in” and “slow out” motions. Moreover, the *Linear* rule is used on the first and last tangent and, lastly, *Smooth* is particularly used on all key frames to automatically adjust tangent for smooth results. We specify *Constant value* extrapolation mode to define how the curve will be extrapolated before the first and after the last key frames.

In Figures (5.20, 5.21, 5.22, 5.23, 5.24) we demonstrate a sequence of snapshots obtained from applying a combination of elementary motions to some of the extracted skeleton morphs. We also provide instances depicting a closer view of a human knee mesh to demonstrate the robustness of the skinning process. These visual results confirm that our method eliminates all the artifacts exhibited by previous LBS methods. However, sometimes defects arise on surfaces with data points that are not uniformly sampled.

As we can see from Tables (5.4, 5.5, 5.6, 5.7), the overall processing time of our animation method is very high for a real-time process due to the computation of new points which takes  $O(n_1 \cdot n_2 \cdot n_3)$  time to complete ( $n_1$  is the number of End points of moving component,  $n_2$  is the number of End points of its father and  $n_3$  is the cardinality of Middle points set) and the triangulation process  $O(n^2)$  complexity, where  $n$  is the number of constructed points. Moreover, increasing the rotation angles results in an increase of the new construction points since the gap splays. Also, as we have mentioned earlier, the vertices construction process depends on the median of the median lengths of the connecting components, since it defines their between distance. We will avoid to post the performance of Keyframing animation and Forward kinematics rigid skinning systems since the time measurements are negligible compared to overall performance. We provide, contrariwise, full performance report of the Gap reconstruction process at following tables, giving processing times of the 4 processes (Remove vertices, Add new vertices, Triangulate vertices, Compute normal vectors) at a number of key frames for each model. These key frames presents moves which impress median and worse processing times for each model. Finally, Table 5.3 shows the pre-processing time (is measured in seconds) of several processes which are essential for our animation framework. A summary of these processes for each component is the computation of its median length, model's components which are not at its sub-tree and points neighborhood.

Table 5.3: Time performance of pre-processing functions. Columns from left to right represent time performance computing median length, components which did not belong to checking node’s sub-tree and neighbors for all vertices.

Model	Median length	Others nodes	Vertex neighborhood
Cow	0.00805	0.00521	1.54492
Homer	0.01023	0.00176	1.99712
Dilo	0.01141	0.00475	2.24194
Camel	0.01247	0.00139	2.40833
Horse	0.01839	0.00595	3.63812
Dino	0.01988	0.00871	3.94615

Time performance of Gap construction system animating a number of components on our model database (measured in seconds). Columns from left to right represent time performance of “removing vertices”, “adding new vertices”, “triangulating vertices” and “computing normal vectors” processes.

- Cow:

Table 5.4: Time performance of Gap construction system animating 4 components of Cow model.

Frames	Remove Vertices	Add Vertices	Triangulate Vertices	Find Normals	Total Time
1	0.04341	0.35256	1.48982	0.03518	1.92097
2	0.04060	0.37861	1.37159	0.03237	1.82317
3	0.04310	0.40203	1.49757	0.03520	1.97790
4	0.03706	0.34560	1.27299	0.02887	1.68452
Mean	0.0410425	0.36970	1.4079925	0.032905	1.85164

- Homer:

Table 5.5: Time performance of Gap construction system animating 4 components of Homer model.

Frames	Remove points	Add points	Triangulate points	Find normals	Total Time
1	0.05949	1.6076	1.98946	0.04805	3.25776
2	0.06120	1.18995	1.94005	0.04551	3.23671
3	0.05424	0.72978	1.83591	0.04293	2.66286
4	0.05394	0.73009	1.84912	0.04351	2.67666
Mean	0.0572175	0.952645	1.903635	0.045	2.9584975

- Dilo:

Table 5.6: Time performance of Gap construction system animating 4 components of Dilo model.

Frames	Remove points	Add points	Triangulate points	Find normals	Total Time
1	0.03873	0.20612	0.98307	0.02103	1.24895
2	0.03523	0.18782	1.00787	0.02187	1.25279
3	0.03230	0.16074	1.02027	0.02023	1.23354
4	0.03497	0.18017	0.99503	0.01994	1.23011
Mean	0.0353075	0.1837125	1.00156	0.0207675	1.2413475

- Camel:

Table 5.7: Time performance of Gap construction system animating 4 components of Camel model.

Frames	Remove points	Add points	Triangulate points	Find normals	Total Time
1	0.03568	0.16972	1.09308	0.02285	1.32133
2	0.03199	0.15799	0.93559	0.01964	1.04521
3	0.03144	0.15324	0.96094	0.02075	1.16637
4	0.03017	0.13707	0.91590	0.01869	1.10183
Mean	0.003232	0.154505	0.9763775	0.0204825	1.158685

Cow Model



Figure 5.20: A sequence of images obtained from applying a variety of motions on Cow model.

Homer Model

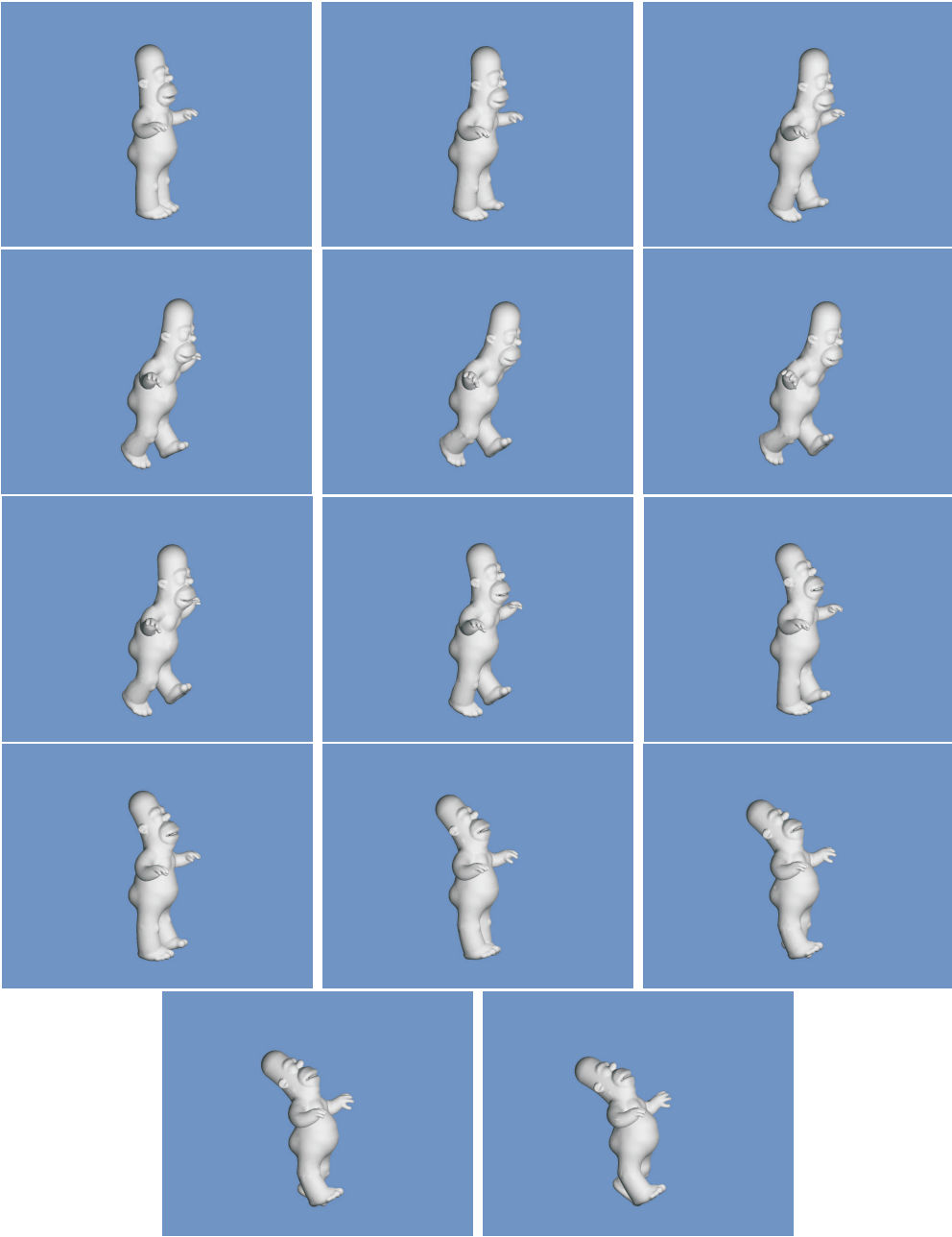


Figure 5.21: A sequence of images obtained from applying a variety of motions on Homer model.



Dilo Model

- Movement 1

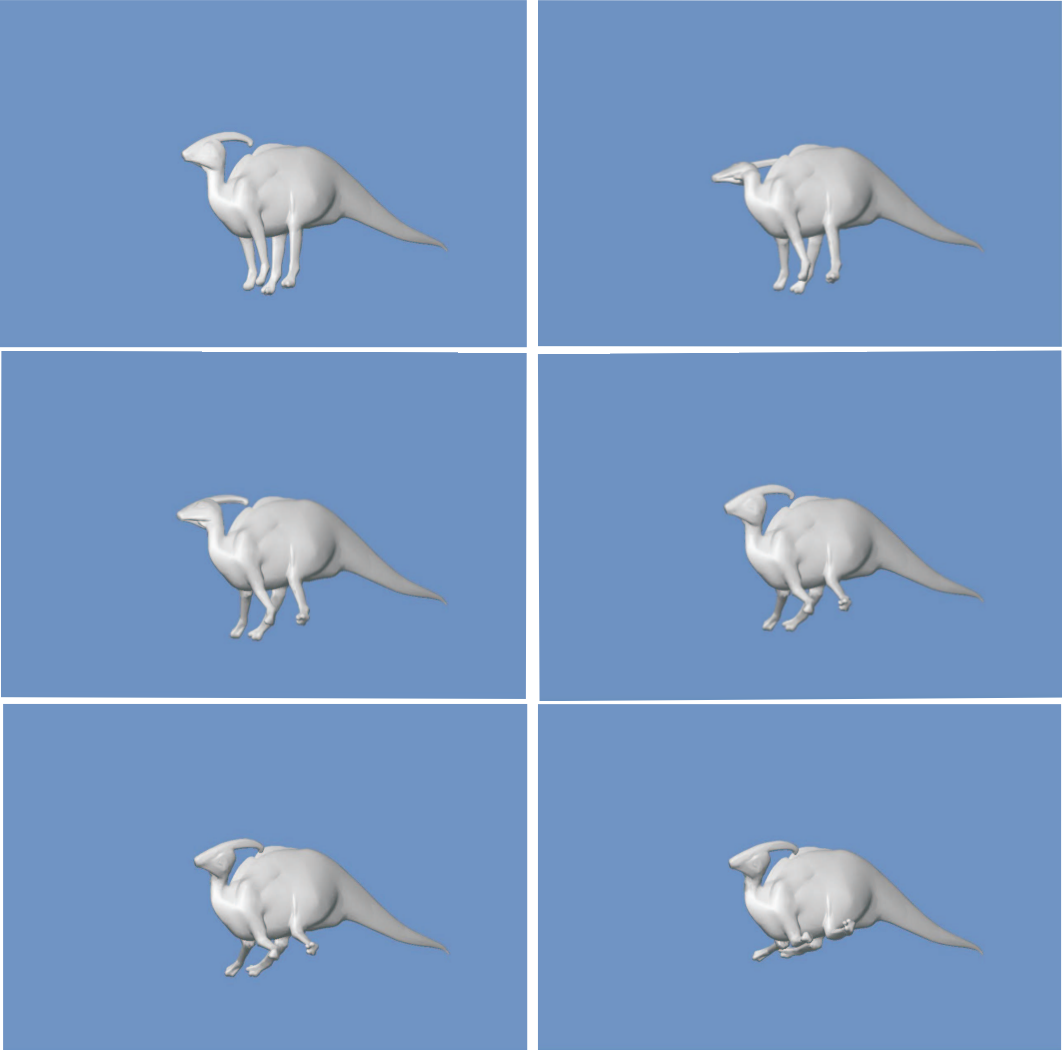


Figure 5.22: First sequence of images obtained from applying a variety of motions on Dilo model.

- Movement 2

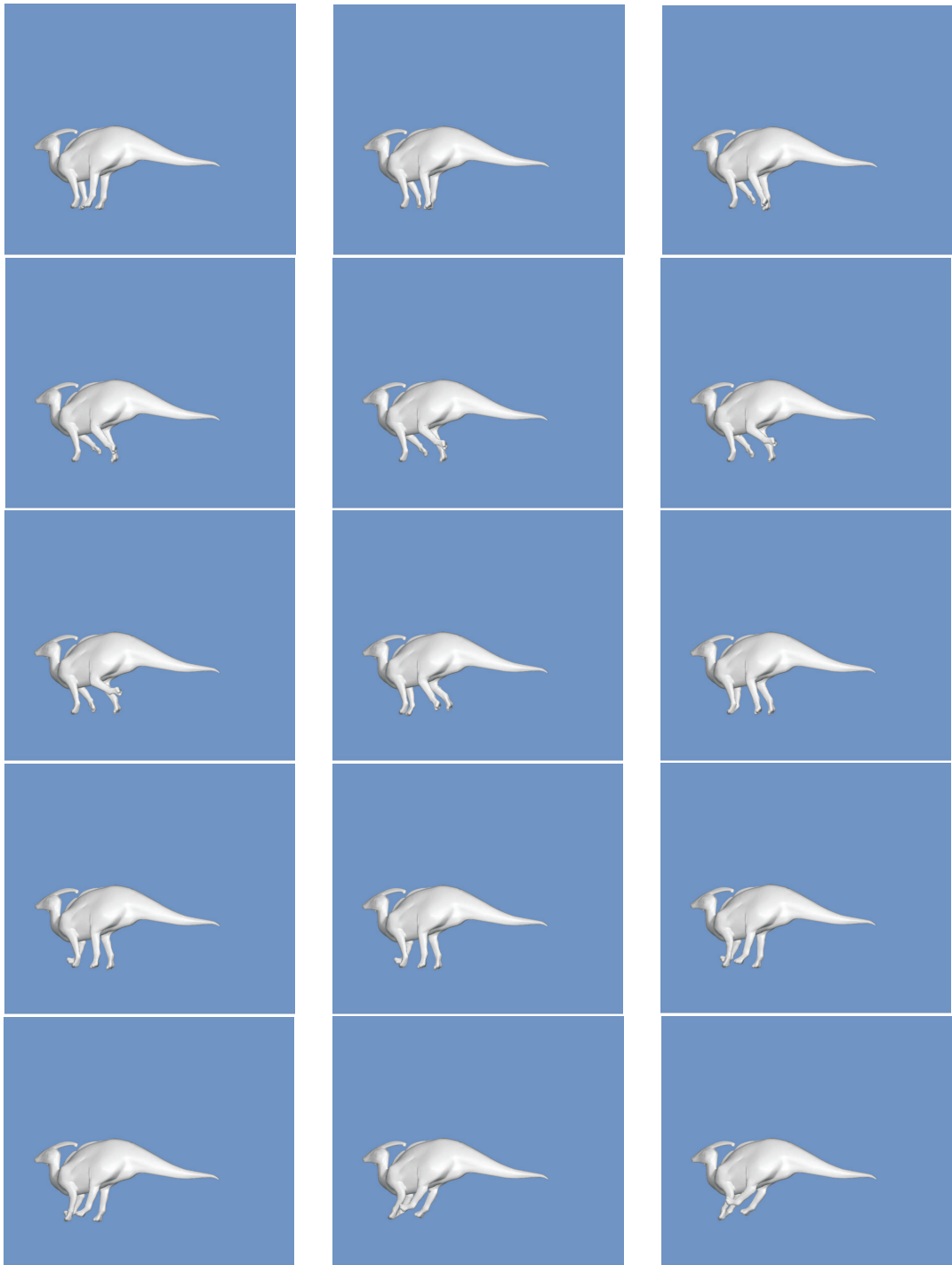




Figure 5.23: Second sequence of images obtained from applying a variety of motions on Dilo model.

Camel Model

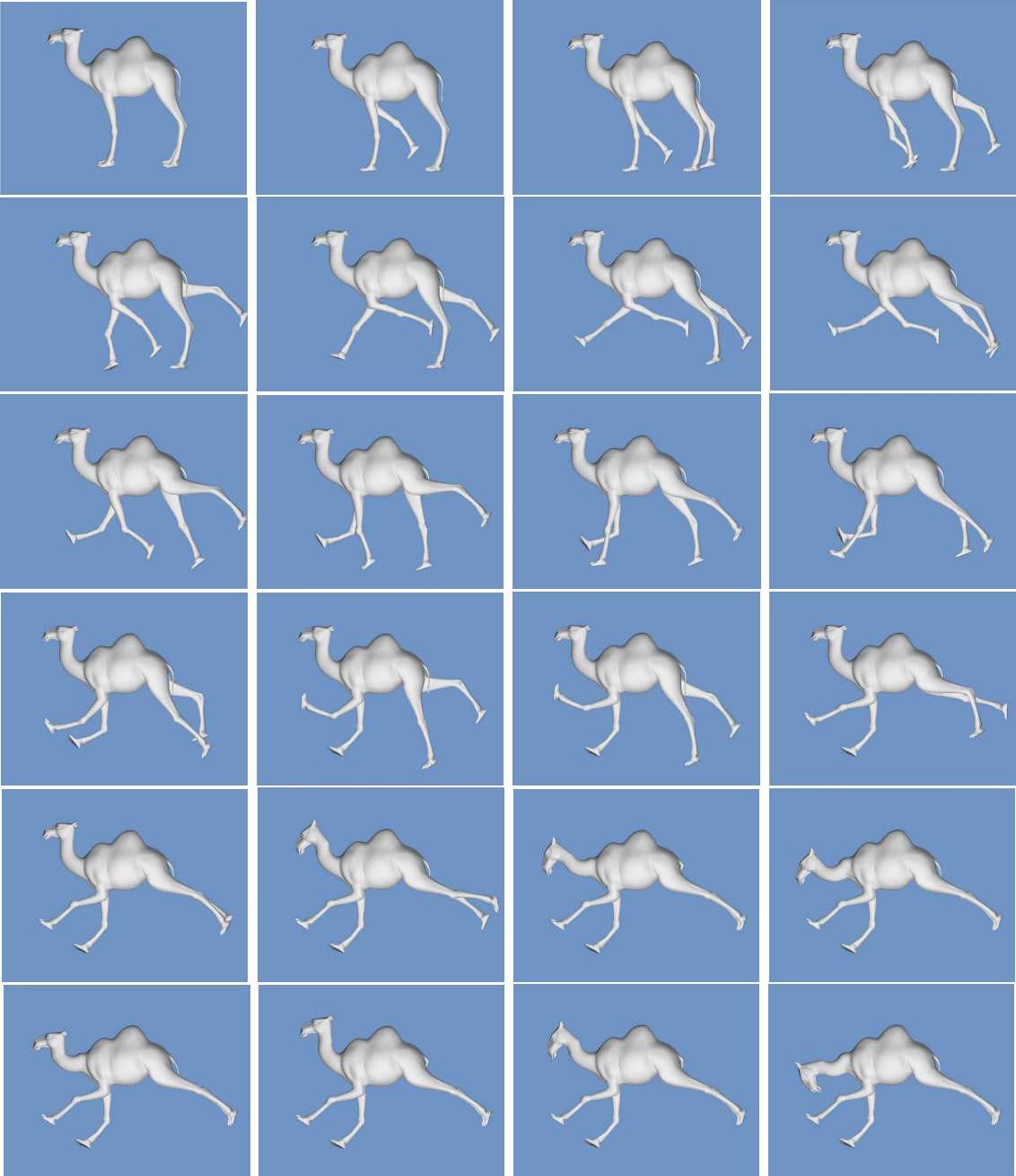


Figure 5.24: A sequence of images obtained from applying a variety of motions on Camel model.

## Skinning Example

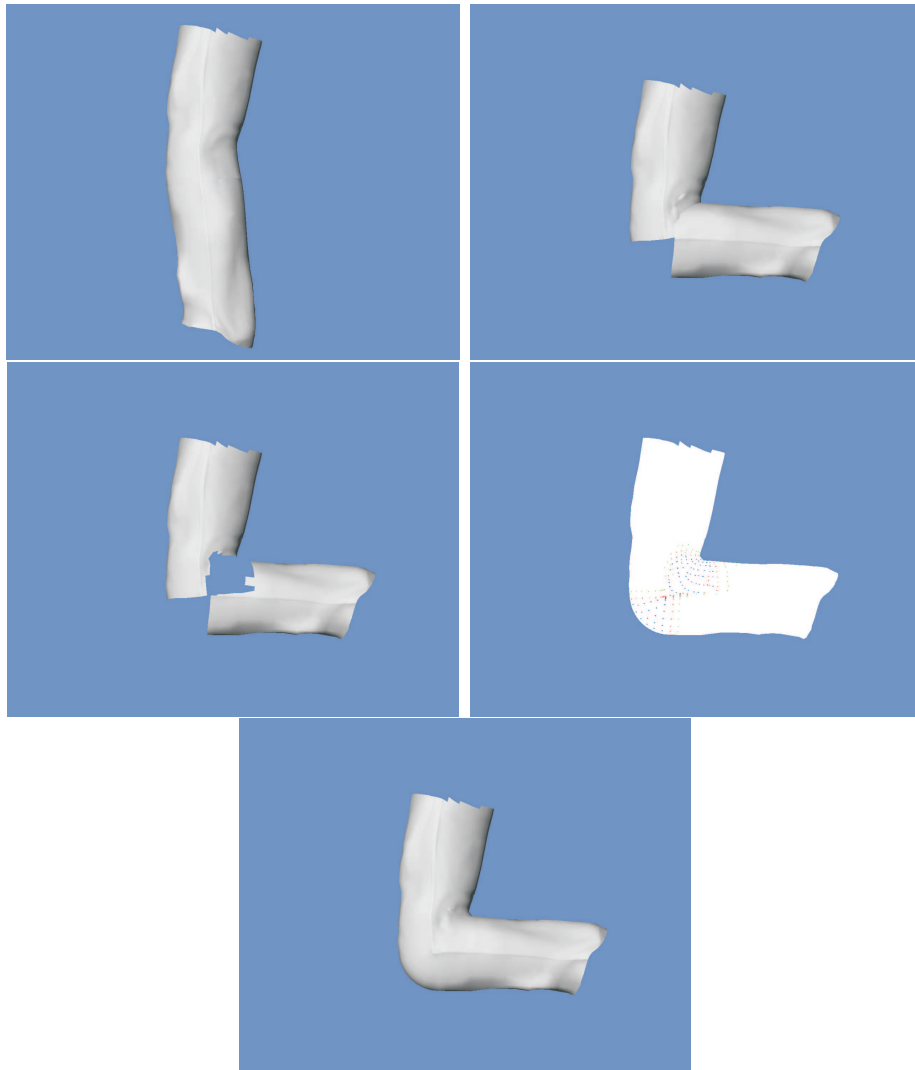


Figure 5.25: A sequence of images describing the steps of the gap reconstruction process. The first picture shows the starting pose of a man's knee. The second one shows the knee's orientation after the use of FK skinning system. The third picture shows how the components are after removing the skin vertices. The fourth picture shows the computed patch vertices (Blue: *Circle*(down) and *Bezier*(up) group, Red: *End* points, Green: *Extra* group ) which reconstruct the gap. Final result (after the triangulation and computation of normals processes completed) is illustrated at the final picture.

## CHAPTER 6

# CONCLUSIONS AND FUTURE WORK

---

In this dissertation, we have studied the problem of skeletal animation of articulated modular solid objects. In this context we have identified the following key issues: the development of efficient multi-resolution skeletonization techniques, the animation of articulation figures through a combination of keyframing and specialized deformation techniques and finally the efficiency and robustness of triangulation methods used for blending solid modules.

All key components of these problems have been studied and various state-of-the-art approaches have been employed to provide effective solutions. We have proposed a robust skeleton-based animation framework for 3D characters. Refined extracted skeletons are used for articulated body animation. We have studied and improved three local skeletonization methods. In addition, we have presented approximate refinement algorithms to improve skeleton's orientation. To avoid vertex weights approximation and sample pose generation cost, we have developed a novel rigid skinning method. This method eliminates the potential shortcomings from self-intersections, providing plausible mesh deformations.

Considering completeness there are many subtasks that could be performed regarding our improved Principal Axis skeletonization technique. There is need for further investigating in a more quantitative manner the grouping functions. Further, one could try replacing the angle-weighted algorithm with an optimization method which will compute the principal axis orientation more accurately.

Also, considering completeness there are some tasks that need to be carried on in our rigid skinning approach. Performing a surface smoothing algorithm in a global way over the triangular patch mesh, could produce better skin results with a slightly increased estimation cost. Finally, to achieve real-time animation an alternative triangulation method (for example CGAL [102] implementation) should be employed and its GPU realization should be investigated.

## BIBLIOGRAPHY

---

- [1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
- [2] M. Alexa, Linear combination of transformations, *ACM Trans. Graph. Volume 21 Number 3* (2002) 380–387.
- [3] B. Allen, B. Culress, Z.Popovic, Articulated Body Deformation From Range Scan Data, *ACM Transactions on Graphics Volume 21 Number 3* (2002) 612–619.
- [4] N. Amenta, M. Bern, M. Kamvysselis, A new Voronoi-based surface reconstruction algorithm, *ACM SIGGRAPH* (1998) 415–421.
- [5] N. Amenta, S. Choi, T.K. Dey, N. Leekha, A simple algorithm for homeomorphic surface reconstruction, *International Journal Computer Geomometry and Applications Volume 12* (2002) 125–141.
- [6] N. Amenta, S. Choi, R.K. Kolluri, The power crust, *Proceeding 6th Annual Symposium Solid Modeling Applications* (2001) 249–260.
- [7] N. Amenta, S. Choi, R.K. Kolluri, The power crust, unions of balls, and the medial axis transform, *Computational Geometry Volume 19 Number 2-3* (2001) 127–153.
- [8] D. Attali, J.O. Lachaud, Delaunay conforming iso-surface; skeleton extraction and noise removal, *Computational Geometry: Theory and Applications Volume 19 Number 2-3* (2001) 175–189.
- [9] M. Attene, S. Biasotti, M. Spagnuolo, Re-Meshing Techniques for Topological Analysis, *SMI '01: Proceedings of the International Conference on Shape Modeling & Applications* (2001) 142.
- [10] M. Attene, S. Katz, M. Mortara, G. Patane, M. Spagnuolo, A. Tal, Mesh Segmentation - A Comparative Study, *SMI '06: Proceedings of the IEEE International Conference on Shape Modeling and Applications 2006* (2006) 7.
- [11] O. K.-C. Au, C.-L. Tai, H.-K. Chu, D. Cohen-Or, T.-Y. Lee, Skeleton Extraction by Mesh Contraction, *ACM Transactions on Graphics Volume 27 Number 3* (2008)

- [12] G. Auja, F. Hétoy, F. Lazarus, C. Depraz, Harmonic skeleton for realistic character animation, *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2007) 151–160.
- [13] M. Attene, S. Biasotti, M. Spagnuolo, Re-meshing techniques for topological analysis, *Proceedings Shape Modeling International* (2001) 142–151.
- [14] C. Bajaj, F. Bernardini, G. Xu, Automatic reconstruction of surfaces and scalar fields from 3D scans, *ACM SIGGRAPH* (1995) 109–118.
- [15] F. Bernardini, J. Mittleman, H. Rushmeier, C. Silva, G. Taubin, The ball-pivoting algorithm for surface reconstruction, *IEEE Transactions on Visual Computer Graphics Volume 5* (1999) 349–359.
- [16] P. Bergeron, P. Lachapelle, Controlling Facial Expression and Body Movements in the Computer Generated Short 'Tony de Peltrie', *ACM SIGGRAPH Tutorial Notes* (1985).
- [17] I. Bitter, A.E. Kaufman, M. Sato, Penalized-distance volumetric skeleton algorithm, *IEEE Transactions on Visualization and Computer Graphics Volume 7 Number 3* (2001) 195–206.
- [18] H. Blum, A transformation for extracting new descriptors of shape, *Models for the Perception of Speech and Visual Form* (1967) 362–380.
- [19] J.D. Boissonnat, Geometric structures for three dimensional shape representation, *ACM Transactions on Graphics Volume 3 Issue 4* (1984) 266–286.
- [20] J.D. Boissonnat, F. Cazals, Smooth surface reconstruction via natural neighbor interpolation of distance functions, *Proceedings 16th Annual Symposium Computer Geometry* (2000) 223–232.
- [21] Cinefex, Riverside, CA, various issues, e.g. vol. 66 (June 1996), p. 52 (Dragonheart); vol. 64 (Dec 1995), p. 62 (Jumanji).
- [22] J.H. Chuang, C.H. Tsai, M.C. Ko, Skeletonization of three-dimensional object using generalized potential field, *IEEE Transactions on Pattern Analysis and Machine Intelligence Volume 22 Number 11* (2000) 1241–1251.
- [23] D. Cohen-Steiner, F. Da, A greedy Delaunay based surface reconstruction algorithm, *The Visual Computer: International Journal of Computer Graphics* (2004) 4–16.
- [24] F. Cordier, N. Magnenat-Thalmann, A Data-Driven Approach for Real-Time Clothes Simulation, *PG '04: Proceedings of the Computer Graphics and Applications, 12th Pacific Conference* (2004) 257–266.



- [25] N.D. Cornea, D. Silver, X.S. Yuan, R. Balasubramanian, Computing hierarchical curve-skeletons of 3D objects, *The Visual Computer Volume 21 Number 11* (2005) 945–955.
- [26] T. Culver, J. Keyser, D. Manocha, Exact computation of the medial axis of a polyhedron, *Computer Aided Geometry Design Volume 21 Number 1* (2004) 65–98.
- [27] B. Curless, M. Levoy. A volumetric method for building complex models from range images, *ACM SIGGRAPH* (1996) 303–312.
- [28] T.K. Dey, S. Goswami, Tight Cocone: A water-tight surface reconstructor, *Journal of computing and Information Science in Engineering Volule 3* (2003) 302–307.
- [29] T.K. Dey, J. Sun, Defining and computing curve-skeletons with medial geodesic function, *SGP '06: Proceedings of the fourth Eurographics symposium on Geometry processing*, (2006) 143–152.
- [30] J. Denavit, R.S. Harteneberg, A kinematic Notation for Lower-pair Mechanisms Based on Matrices, *Journal of Applied Mechanics Volume 23* (1955) 215–221.
- [31] T.K. Dey, W. Zhao, Approximate medial axis as a voronoi subcomplex, *ACM Symposium on Solid Modeling and Applications* (2002) 356–366.
- [32] H. Edelsbrunner, Surface reconstruction by wrapping finite point set in space. (2003) 379–404.
- [33] H. Edelsbrunner and E.P. Mucke, Three-dimensional alpha shapes, *ACM Transactions on Graphics Volume 13* (1994) 43–72.
- [34] S. Forstmann, J. Ohya, A. Krohn-Grimberghe, R. McDougall, Deformation styles for spline-based skeletal animation, *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2007) 141–150.
- [35] I. Fudos, C.M. Hoffmann, Constraint-based parametric conics for CAD, *Computer-Aided Design Volume 28 Number 2* (1996) 91–100.
- [36] S. Funke, E.A. Ramos, Smooth-surface reconstruction in near-linear time, *Proceeding 13th ACM-SIAM Annual Symposium Discrete Algorithms* (2002) 781–790.
- [37] N. Gagvani, D. Silver, Animating volumetric models, *Graph. Models Volume 63 Number 6* (2001) 443–458.
- [38] J. Giesen, M. John, The Flow complex: A data structure for geometric modeling, *Proceedings 14th. ACM-SIAM Annual Symposium Discrete Algorithms* (2003) 285–294.

- [39] M. Gopi, S. Krishnan, C.T. Silva, Surface reconstruction based on lower dimensional localized delaunay triangulation, *Computer Graphics Forum Volume 19 Issue 3* (2002) 467–478.
- [40] S. Gottschalk, Collision Queries Using Oriented Bounding Boxes, Ph.D. Thesis, Department of Computer Science, University of North Carolina, Chapel Hill, 2000.
- [41] S. Gottschalk, M.C. Lin, D. Manocha, OBBTree: A hierarchical Structure for Rapid Interference Detection, *ACM SIGGRAPH* (1996) 171–180.
- [42] Sir W.R.Hamilton, *Elements of Quaternions, Third Edition*, Chelsea Publishing Co., 1963.
- [43] J. Hejl Hardware skinning with quaternions, *Game Programming Gems Volume 4* (2004) 487–495.
- [44] M. Hilaga, Y. Shinagawa, T. Kohmura, T. L. Kunii, Topology matching for fully automatic similarity estimation of 3D shapes, *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001) 203–212.
- [45] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, W. Stuetzle, Surface reconstruction from unorganized points, *ACM SIGGRAPH* (1992) 71–78.
- [46] W.M. Hsu, J.F. Hughes, H. Kaufman, Direct Manipulation of Free-Form Deformations, *Computer Graphics Volume 26 (Proceedings SIGGRAPH)* (1992) 177–184.
- [47] D.-E. Hyun, S.-H. Yoon, J.-W. Chang, J.-K. Seong, M.-S. Kim, B. Juttler, Sweep-based Human Deformation, *The Visual Computer Volume 21 Number 8–10* (2005).
- [48] D. Jacka, A. Reid, B. Merry, J.Gain, A comparison of linear skinning techniques for character animation, *AFRIGRAPH '07: Proceedings of the 5th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa* (2007) 177–186.
- [49] D.L. James, D.K. Pai, Skinning Mesh Animations, *ACM Transactions on Graphics Volume 24 Number 3* (2005) 399–407.
- [50] S. Katz, A. Tal, Hierarchical mesh decomposition using fuzzy clustering and cuts, *ACM Transactions on Graphics Volume 22 Number 3* (2003) 954–961.
- [51] L. Kavan, J. Žára, Spherical blend skinning: a real-time deformation of articulated models, *I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games* (2005) 9–16.
- [52] L. Kavan, S. Collins, J. Zara, C. O'Sullivan, Geometric Skinning with Approximate Dual Quaternion Blending, *ACM Trans. Graph. Volume 27 Number 4* (2008).

- [53] P. G. Kry, D. L. James, D. K. Pai, EigenSkin: real time large deformation character skinning in hardware, *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2002) 153–159.
- [54] K. Komatsu, Human Skin Model Capable of Natural Shape Variation, *The Visual Computer Volume 4 Number 3* (1988) 265–271.
- [55] Y.K. Lai, S.M. Hu, R.R. Martin, P.L. Rosin, Fast Mesh Segmentation using Random Walks, *ACM Symposium on Solid and Physical Modeling* (2008) 183–191.
- [56] J. Lander, Skin them bones: Game programming for the web generation, *Game Developer Magazine* (1998) 11–16.
- [57] J. Lander, Over my dead, polygonal body, *Game Developer Magazine* (1999) 17–22.
- [58] D.T. Lee, F.P. Preparata, An Optimal Algorithm for Finding the Kernel of a Polygon, *Journal of the ACM, Volume 26, Issue 3* (1979) 415–421.
- [59] J.P. Lewis, M. Cordner, N. Fong, Pose Space Deformations: A Unified Approach to Shape Interpolation and Skeleton-Driven Deformation, *ACM SIGGRAPH Proceedings, Computer Graphics Proceedings, Annual Conference Series* (2000) 165–172.
- [60] X. Li, T.W. Woon, T.S. Tan, Z. Huang, Decomposing polygon meshes for interactive applications, *In Proceeding of the 2001 symposium on Interactive 3D graphics* (2001) 35–42
- [61] J.M. Lien, N.M. Amato, Approximate convex decomposition of polyhedra, *ACM Symposium on Solid and Physical Modeling* (2007) 121–131.
- [62] M. Lee, Seven ways to skin a mesh: Character skinning revisited for modern GPUs, *In Proceedings of GameFest, Microsoft Game Technology Conference* (2006).
- [63] J.M. Lien, J. Keyser, N.M. Amato, Simultaneous Shape Decomposition and Skeletonization, *In Proc. ACM Solid and Physical Modeling Symposium (SPM)* (2006) 219–228.
- [64] P.-C. Liu, F.C. Wu, W.-C. Ma, R.-H. Liang, M. Ouhyoung, Automatic animation skeleton using repulsive force field, *Computer Graphics and Applications, 2003. Proceedings. 11th Pacific Conference on* (2003) 409–413
- [65] C.-M. Ma, S.-Y. Wan, J.-D. Lee, Three-Dimensional Topology Preserving Reduction on the 4-Subfields, *IEEE Trans. Pattern Anal. Mach. Intell. Volume 24 Number 12* (2002) 1594–1605.
- [66] G. Maestri, *Digital Character Animation 2, Volume 1*, New Rider, 1999.

- [67] N. Magnenat-Thalmann, R. Laperriere, D. Thalmann, Joint-Dependent Local Deformations for Hand Animation and Object Grasping, *Proceedings Graphics Interface* (1988) 26–33.
- [68] N. Magnenat-Thalmann, F. Cordier, H. Seo, d G. Papagianakis, Modeling of Bodies and Clothes for Virtual Environments, *CW '04: Proceedings of the 2004 International Conference on Cyberworlds* (2004) 201–208.
- [69] B. Merry, P. Marais, J. Gain, Animation space: A truly linear framework for character animation, *ACM Trans. Graph. Volume 25, Number 4* (2006) 1400–1423.
- [70] A. Mohr, M. Gleicher, Building Efficient, Accurate Character Skins From Examples, *ACM Transactions on Graphics Volume 22 Number 3* (2003) 562–568.
- [71] R.M. Palenichka, M.B. Zaremba, Multi-scale model-based skeletonization of object shapes using self-organizing maps, *International Conference on Pattern Recognition (ICPR) Volume 1* (2002) 10143–10147.
- [72] F.I. Parke, Parameterized Models for Facial Animation. *IEEE Computer Graphics and Applications Volume 2 Number 9* (1982) 61–68.
- [73] V. Pascucci, G. Scorzelli, P.-T. Bremer, A. Mascarenhas, Robust on-line computation of Reeb graphs: simplicity and speed, *ACM Transactions on Graphics Volume 26 Number 3* (2007) 58.
- [74] T. Rhee, J.P. Lewis, U. Neumann, Real-time weighted pose-space deformation on the GPU, *Computer Graphics Forum Volume 25 Number 3* (2006) 439–448.
- [75] S. Schaefer, C. Yuksel, Example-based skeleton extraction, *SGP '07: Proceedings of the fifth Eurographics symposium on Geometry processing* (2007) 153–162.
- [76] E. C. Sherbrooke, N. M. Patrikalakis, E. Brisson, An Algorithm for the Medial Axis Transform of 3D Polyhedral Solids, *IEEE Transactions on Visualization and Computer Graphics Volume 2 Number 1* (1996) 44–61.
- [77] Y. Shinagawa, T.L. Kunii, Y.L. Kergosien, Surface coding based on morse theory, *IEEE Computer Graphics and Applications Volume 11 Number 5* (1991) 66–78.
- [78] K. Siddiqi, A. Shokoufandeh, S.J. Dickinson, S.W. Zucker, Shock graphs and shape matching, *ICCV* (1998) 222–229.
- [79] K. Sims, D. Zeltzer, A figure Editor and Gait Controller for Task Level Animation, *SIGGRAPH Course Notes Number 4* (1988) 164–181.
- [80] P.-P. J. Sloan, C. F. Rose, III, M. F. Cohen, Shape by example, *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics* (2001) 135–143.

- [81] G. Turk, M. Levoy, Zippered polygon meshes from range images, *Proceedings SIG-GRAPH* (1994) 311–318.
- [82] A. Verroust, F. Lazarus, Extracting skeletal curves from 3d scattered data, *International Conference on Shape Modeling and Applications. IEEE Computer Society* (1999) 194–201.
- [83] L. Vicci, Quaternions and Rotations in 3-Space: The Algebra and its Geometric Interpretation, *Technical Report TR01-014, Dept. of Computer Science, University of North Carolina at Chaper Hill* (2001).
- [84] L. Wade, R.E. Parent, Automated Generation of Control Skeletons for Use in Animation, *The Visual Computer Volume 18* (2000).
- [85] , X. C. Wang, C. Phillips, Multi-weight enveloping: least-squares approximation techniques for skin animation, *SCA '02: Proceedings of the 2002 ACM SIG-GRAPH/Eurographics symposium on Computer animation* (2002) 129–138.
- [86] R. Y. Wang, K. Pulli, J. Popović, Real-time enveloping with rotational regression, *ACM Trans. Graph. Volume 26 Number 3* (2007) 73.
- [87] J. Weber, Run-time skin deformation, *In Proceedings of Game Developers Conference* (2000)
- [88] O. Weber, O. Sorkine, Y. Lipman, C. Gotsman, Context-Aware Skeletal Shape Deformation, *Computer Graphics Forum (Proceedings of Eurographics) Volume 26 Number 3* (2007) 265–274.
- [89] F.C. Wu, W.C. Ma, P.C. Liou, R.H. Laing, M. Ouhyoung, Skeleton extraction of 3d objects with visible repulsive force, *Computer Graphics Workshop* (2003).
- [90] X. Yang, A. Somasekharan, J.J. Zhang, Curve skeleton skinning for human and creature characters: Research Articles, *Comput. Animat. Virtual Worlds Volume 17 Number 3–4* (2006) 281–292.
- [91] S.-H. Yoon, M.-S. Kim, Sweep-based Freeform Deformations, *Computer Graphics Forum Volume 25 Number 3* (2006) 487–496.
- [92] Y. Zhou, A.W. Toga, Efficient skeletonization of volumetric objects, *IEEE Transactions on Visualization and Computer Graphics Volume 5 Number 3* (1999) 196–209.
- [93] C. Eriscon, *Real Time Collisison Detection*, Morgan Kaufmann, 2004.
- [94] F.S. Hill Jr, *Computer Graphics using OpenGL, Second Edition*, Prentice Hall, 2000.
- [95] F. Preparata, M. Shamos, *Computational Geometry*, Springer-Verlag, 1985.

- [96] P. Shirley, M. Ashikhmin, M. Gleicher, S. Marschner, E. Reinhard, K. Sung, W. Thompson, P. Willemsen, *Fundamentals of Computer Graphics, Second Edition*, A K Peters, 2005.
- [97] A. Watt, F. Policarpo, *3D Games, Real-time rendering and software technology: Volume 1, 2*, Addison-Wesley, 2000.
- [98] A. Watt, M. Watt, *Advanced Animation and Rendering Techniques - Theory and Practice*, Addison-Wesley, 1992.
- [99] <http://www.autodesk.com/3dsmax/>
- [100] <http://www.blender.org/>
- [101] <http://www.devmaster.net/>
- [102] <http://www.cgal.org/>
- [103] <http://www.gamasutra.com/>
- [104] <http://www.geometrictools.com/>
- [105] <http://www.opengl.org/resources/libraries/glut/>
- [106] <http://www.netlib.org/lapack/>
- [107] <http://www.pixar.com/>
- [108] <http://www.qhull.org/>
- [109] <http://shapes.aim-at-shape.net/>
- [110] <http://en.wikipedia.org/>

# APPENDIX

---

## Linear Algebra

### A. Linear Systems Solutions

A system of *linear equations* is a collection of linear equations involving the same set of variables. A solution to a linear system is an assignment of numbers to the variables such that all the equations are simultaneously satisfied. A general system of  $m$  linear equations with  $n$  unknowns can be written as

$$\begin{array}{cccccc} a_{11}x_1 & + & a_{12}x_2 & + & \cdots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \cdots & + & a_{2n}x_n & = & b_2 \\ \vdots & & \vdots & & & & \vdots & & \vdots \\ a_{m1}x_1 & + & a_{m2}x_2 & + & \cdots & + & a_{mn}x_n & = & b_m \end{array}$$

Here  $x_1, x_2, \dots, x_n$  are the unknowns,  $a_{11}, a_{12}, \dots, a_{mn}$  are the coefficients of the system, and  $b_1, b_2, \dots, b_m$  are the constant terms. This equation is equivalent to a matrix form equation  $Ax = b$ , where  $A$  is an  $m \times n$  matrix,  $x$  is a column vector with  $n$  entries, and  $b$  is a column vector with  $m$  entries,

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

There are several algorithms for solving a system of linear equations. We choose to use the Gauss Elimination method.

## A.1 Gauss Elimination method

The process of Gaussian elimination has two parts. The first part (*Forward Elimination*) reduces a given system to triangular form  $Ux = y$ , where  $U$  upper triangle matrix or results in a degenerate equation with no solution, indicating the system has no solution. This is accomplished through the use of elementary row operations (multiplying rows, switching rows, and adding multiples of rows to other rows). Now, the linear system has the form:

$$\left\{ \begin{array}{l} u_{11}x_1 + u_{12}x_2 + \cdots + u_{1n}x_n = y_1 \\ \phantom{u_{11}x_1} + u_{12}x_2 + \cdots + u_{1n}x_n = y_2 \\ \phantom{u_{11}x_1} \phantom{+ u_{12}x_2} + \cdots + u_{1n}x_n = y_3 \\ \phantom{u_{11}x_1} \phantom{+ u_{12}x_2} \phantom{+ \cdots + u_{1n}x_n} = y_m \end{array} \right.$$

which can be easily solved by *back substitution*. Firstly, we solve the last equation for  $x_n$ , replace  $x_n$  on the penultimate equation and solve this for  $x_{n-1}$ , etc, and finally replace  $x_2, x_3, \dots, x_n$  on the first equation, solving for  $x_1$ .

## B. Calculating Eigenvalues/Eigenvectors

Given a linear transformation  $A$ , a non-zero vector  $x$  is defined to be an *eigenvector* of the transformation if it satisfies the eigenvalue equation  $Ax = \lambda x$  for some scalar  $\lambda$ . Namely, the eigenvectors are those non-zero vectors whose directions do not change when multiplied by the matrix. In this situation, the scalar  $\lambda$  is called an *eigenvalue* of  $A$  corresponding to the eigenvector  $x$ . There are several algorithms for calculating eigenvalues and eigenvectors of linear transformation. We choose to use the Singular Value Decomposition method.

### B.1 Singular Value Decomposition method

Suppose  $M$  is an  $m \times n$  matrix whose entries come from the field  $K$ , which is either the field of real numbers or the field of complex numbers. Then there exists a factorization of the form  $M = U\Sigma V^*$ , where  $U$  is an  $m \times m$  unitary matrix over  $K$ , the matrix  $\Sigma$  is  $m \times n$  with non-negative numbers on the diagonal (as defined for a rectangular matrix) and zeros off the diagonal, and  $V^*$  denotes the conjugate transpose of  $V$ , an  $n \times n$  unitary matrix over  $K$ . Such a factorization is called a *singular-value decomposition* of  $M$ .

- The matrix  $V$  thus contains a set of orthonormal "input" or "analyzing" basis vector directions for  $M$
- The matrix  $U$  contains a set of orthonormal "output" basis vector directions for  $M$



- The matrix  $\Sigma$  contains the singular values, which can be thought of as scalar "gain controls" by which each corresponding input is multiplied to give a corresponding output.

The singular value decomposition is very general in the sense that it can be applied to any  $m \times n$  matrix. The eigenvalue decomposition, on the other hand, can only be applied to certain classes of square matrices. Nevertheless, the two decompositions are related. Given an SVD of  $M$ , as described above, the following two relations hold:

$$M^*M = V\Sigma^*U^*U\Sigma V^* = V(\Sigma^*\Sigma)V^*$$

$$MM^* = U\Sigma V^*V\Sigma^*U^* = U(\Sigma^*\Sigma)U^*$$

The right hand sides of these relations describe the eigenvalue decompositions of the left hand sides. Consequently, the squares of the non-zero singular values of  $M$  are equal to the non-zero eigenvalues of either  $M^*M$  or  $MM^*$ . Furthermore, the columns of  $U$  (left singular vectors) are eigenvectors of  $MM^*$  and the columns of  $V$  (right singular vectors) are eigenvectors of  $M^*M$ .

## C. Principal Component Analysis

*Principal component analysis (PCA)* is mathematically defined as an orthogonal linear transformation technique that used to reduce multidimensional data sets to lower dimensions such that the greatest variance by any projection of the data comes to lie on the first coordinate (called the first principal component), the second greatest variance on the second coordinate, and so on. PCA can be used for dimensionality reduction in a data set by retaining those characteristics of the data set that contribute most to its variance, by keeping lower-order principal components and ignoring higher-order ones. Such low-order components often contain the "most important" aspects of the data. However, depending on the application this may not always be the case. PCA involves the calculation of the eigenvalue decomposition of a data covariance matrix or singular value decomposition of a data matrix, usually after mean centering the data for each attribute. Following is a detailed description of PCA using the covariance method. The goal is to transform a given data set  $X$  of dimension  $M$  to an alternative data set  $Y$  of smaller dimension  $L$ .

- Organize the data set

Suppose you have data comprising a set of observations of  $M$  variables, and you want to reduce the data so that each observation can be described with only  $L$  variables,  $L < M$ . Suppose further, that the data are arranged as a set of  $N$  data vectors  $X_1, \dots, X_n$  with each  $X_i$  representing a single grouped observation of the  $M$  variables. Write  $X_1, \dots, X_n$  as column vectors, each of which has  $M$  rows and place them into a single matrix  $X$  of dimensions  $M \times N$ .

- Calculate the empirical mean

Firstly, we have to find the empirical mean along each dimension  $m = 1, \dots, M$ . This can be done by placing the calculated mean values into an empirical mean vector  $u$  of dimensions  $M \times 1$ .

$$u[m] = \frac{1}{N} \sum_{n=1}^N \mathbf{X}[m, n]$$

- Calculate the deviations from the mean

Then, we subtract the empirical mean vector  $u$  from each column of the data matrix  $X$  and store the mean-subtracted data in the  $M \times N$  matrix  $B$ .

$$\mathbf{B} = \mathbf{X} - u \cdot h,$$

where  $h$  is a  $1 \times N$  row vector of all 1's:  $h[i] = 1$  for  $i = 1, \dots, n$ .

- Find the covariance matrix

In this step, we must find the  $M \times M$  empirical covariance matrix  $C$  from the outer product of matrix  $B$  with itself:

$$\mathbf{C} = \mathbb{E}[\mathbf{C} \otimes \mathbf{C}] = \mathbb{E}[\mathbf{C} \cdot \mathbf{C}^*] = \frac{1}{N} \mathbf{C} \cdot \mathbf{C}^*,$$

- Find the eigenvectors and eigenvalues of the covariance matrix

In order to calculate eigenvalues and eigenvectors of the covariance matrix, we choose to use the Singular value decomposition method as mention above. After the SVD procedure produce the solution we notice that the eigenvalues and eigenvectors are ordered and paired (The  $m$ th eigenvalue corresponds to the  $m$ th eigenvector) but not sorted.

- Rearrange the eigenvectors and eigenvalues

Finally, we have to sort the columns of the eigenvector matrix  $V$  and eigenvalue matrix  $D$  in order of decreasing eigenvalue.

## SHORT VITA

---

Andreas-Alexandros Vasilakis was born on October 12, 1983, in Corfu. He graduated from the 2nd High School of the same city and was admitted for pursuing undergraduate studies at the Computer Science Department of University of Ioannina. He received his B.S degree in 2006 and is currently attending at the post-graduate programme of the same department.