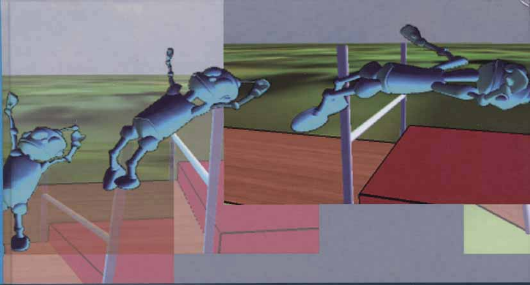


Parag Chaudhuri · Prem Kalra · Subhashis Banerjee

View-Dependent Character Animation

Springer

View-Dependent Character Animation

Parag Chaudhuri, Prem Kalra
and Subhashis Banerjee

View-Dependent Character Animation

Springer

Parag Chaudhuri
Prem Kalra
Subhashis Banerjee
Department of Computer Science and Engineering
Indian Institute of Technology Delhi
New Delhi, India

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Control Number: 2006934623

ISBN-10: 1-84628-591-7 eISBN-10: 1-84628-762-6
ISBN-13: 978-1-84628-591-2 eISBN-13: 978-1-84628-762-6

Printed on acid-free paper

© Springer-Verlag London Limited 2007
Apart from any fair dealing for the purposes of research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act 1988, this publication may only be reproduced, stored or transmitted, in any form or by any means, with the prior permission in writing of the publishers, or in the case of reprographic reproduction in accordance with the terms of licences issued by the Copyright Licensing Agency. Enquiries concerning reproduction outside those terms should be sent to the publishers.

The use of registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant laws and regulations and therefore free for general use.

The publisher makes no representation, express or implied, with regard to the accuracy of the information contained in this book and cannot accept any legal responsibility or liability for any errors or omissions that may be made.

9 8 7 6 5 4 3 2 1

springer.com

Acknowledgments

The authors want to thank Ashwani Jindal for being an extremely helpful collaborator. A special thanks to Vaishali Paithankar for helping with generation of a video used as an example in this book. The authors have also benefited from comments and suggestions received from the following people: Professor Sharat Chandran, Professor Karan Singh, Subhajit Sanyal, and Ayesha Choudhary.

The authors are also thankful to the extremely enthusiastic, helpful, and cooperative staff of Springer, particularly Helen Desmond, Beverley Ford, and Alfred Hofmann.

The authors wish to acknowledge the support of Mr. K. R. Kaushik in making the Vision and Graphics Lab a fantastic place for doing research. The authors want to thank the faculty, staff, and their many friends from the Department of Computer Science and Engineering for their help and support.

Finally, the authors wish to express their gratitude to their family members for their constant encouragement and patience.

Indian Institue of Technology Delhi,
India

Parag Chaudhuri
Prem Kalra
Subhashis Banerjee

Preface

Animation has grown immensely over the years to become a mainstream art form in the hugely active industries of motion films, television, and advertising. A few basic rules and principles about how to harness the technology that gives the illusion of life to still drawings or objects and how to string together individual shots and scenes to tell a story or create a homogeneous, meaningful sequence or mood govern the fundamental aspects of creating an animation.

Computer animation builds upon these fundamentals and uses computer - generated imagery (CGI) to weave magic on the screen. In spite of the fact that the technology employed in creating animation has advanced by leaps and bounds over the years, animation remains a very labourious process, involving a lot of skill and often many iterations, before the magic looks just right. This is the reason why computer animation remains a very active area for research.

Animations where the character and the rendering camera both move are known as *moving-camera character animations*. The sheer number of parameters the animator has to control, in order to get the desired action shot from the intended camera position, is overwhelming. We present, in this book, view-dependent animation as a solution to the challenges encountered during the creation of moving-camera character animations.

Creation of 3D character animations in which the viewpoint changes in every frame is a challenging problem because it demands a definite relation to be preserved between the character and the camera, in order to achieve clarity in staging. We present view-dependent animation as a solution to this arduous problem. In view-dependent animation, the character's pose *depends* on the view. The camera and character pose association, once specified by the animator, is maintained automatically throughout such an animation. We design a general framework to create view-dependent animations.

We formulate the concept of a view space of key views and associated key character poses. The view space representation captures all the information contained in a camera matrix, i.e., the position of the camera center, the direction of viewing, and the focal length of the camera, concisely and elegantly. Any camera path traced on the envelope of this view space generates a view-dependent animation. This facilitates fast and easy exploration of the view space in terms of the view-dependent animations it can generate.

We present a pipeline to create the view space from sketches and a base three-dimensional (3D) mesh model of the character to be animated. Robust computer vision techniques are used to recover the camera from the sketches. We present two novel view-dependent algorithms, which allow us to embed a multilayered deformation system into a view-dependent setting and integrate it with computer vision techniques. These algorithms match the pose of the 3D character to the sketched pose. The recovered camera and pose form the key views and key character poses and create a view space that can be used to generate a view-dependent animation by tracing paths on it.

We then analyze the problem of authoring view-dependent animations from multimodal inputs. We demonstrate that we can extract the relevant information about the cameras and character poses from a video sequence and create a view space. The view space serves as a common representation for all the information contained in different input types like sketches, video, and motion capture. Hence, it is used to integrate all these inputs together. We show that we can use this combined information to generate a view-dependent animation in real time as the animator traces a path on the view space.

We introduce the concept of *stylistic reuse* and formulate it in terms of our framework. We present three techniques for reusing camera-controlled pose variations to animate multiple view-dependent instances of the same character, a group of distinct characters, or the body parts of the same character.

The book is addressed to a broad audience. It should be of great value to both practitioners and researchers in the area of computer animation. We also cover all the prior work relevant to the topics presented in this book so that there is no specific prerequisite. A basic familiarity with the area of computer animation and computer graphics should be sufficient. The book has a lot of figures to help understand all the concepts introduced in it. We have included an example animation for every facet of view-dependent character animation we have explored in this book. All the example animations are available at http://www.cse.iitd.ernet.in/~parag/vdabook.

We would very much appreciate receiving comments and suggestions from the readers.

Indian Institute of Technology Delhi,
India

Parag Chaudhuri
Prem Kalra
Subhashis Banerjee

Contents

1 Introduction

The word *animate* literally means "*to give life to*." Animation can be thought of as the process of making objects move and creating an *illusion of life* [124]. The animator is the person who directs and composes this movement. Since movements of objects and creatures in an animation are generally inspired by how they move in real life, animation is easy, in principle. But as the famous Disney animator Bill Tytla once said, "There is no particular mystery in animation... it's really very simple, and like anything that is simple, it is about the hardest thing in the world to do."

Traditionally, animation began with each frame being painted by hand and then filmed. Over the course of many years animators perfected the ability to impart unique, endearing personalities to their characters. Many technical developments including the introduction of colour and sound, the use of translucent cels (short for celluloid) in compositing multiple layers of drawings into a final image, and the Disney multiplane camera [124] helped animation mature into a rich and complex art form.

Computer animation is the modern day avatar of animation where the computer is used to draw (or render) the moving images. With the advent of the computer, animation has gradually moved into the realm of three dimensions. The computer is primarily used as a tool to interact with the characters in 3D in order to define and control their movement. Today, animated characters span across a diverse spectrum ranging from cartoonlike humans (*The Incredibles* [15]) to fantasy characters (*Shrek* [1]) and from animals (*Madagascar* [35]) to photorealistic humans (*Final Fantasy: The Spirits Within* [114]). The need to animate such diverse characters has caused character animation to become an extensively researched area.

Coordinating and presenting the character's movement in three dimensions to convey a specific idea to the audience, however, still remains an arduous challenge. The animator has to employ a lot of artistic and technical skill, and often a labourious iterative trial-and-error process to achieve a desired combination of the character's action and the point of view from which it is shown. Since in computer animation, values of many parameters that govern the appearance and the movement of the character, can be varied, the animator has an overwhelming number of things to control. It is especially difficult for the animator to generate the character's action if the point of view (i.e., the rendering camera) is also moving. This book deals specifically with the problem of creating moving-camera character animations using a technique called *view-dependent character animation.*

Creating moving-camera character animations in three dimensions is a multi-faceted computer graphics and computer vision problem. It warrants a formal representation of the moving camera and efficient algorithms to help author the multitude of character poses required for the animation. One also has to deal with issues pertaining to camera, character pose interpolation, and visualization of the association between the two. Therefore, the solution to this problem, on one hand, has to be efficient and elegant from the perspective of a computer scientist. On the other hand, the solution must make sense and be intuitive to use for the animator. We develop and demonstrate a framework in an endeavour to find such a solution.

To set the context for developing a framework for moving-camera character animation, it is important to understand the fundamental principles behind animation. This chapter discusses the animation pipeline and draws inspiration from well established animation practices to introduce the idea of view-dependent character animation.

1.1 Principles of Animation

The primary aim of animation is to infuse life into characters. This required the early practitioners of animation to experiment with a plethora of methods for depicting movement on paper. In order to perfect this art, early animators who made sketches of moving human figures and animals, studied models in motion as well as live action film, playing certain actions over and over. The analysis of action became important to the development of animation. The animators continually searched for better ways to communicate the lessons they learned. Gradually, procedures were isolated and named, analyzed and perfected, and new artists were taught these practices as rules of the trade. These came to be known as the *principles of animation* [124]. These principles are

1. *Squash and stretch* – Defining the rigidity and mass of an object by distorting its shape during an action.
2. *Timing* – The spacing of actions in time to define the weight and size of objects, and the personality of characters.
3. *Anticipation* – The preparation for an upcoming action so that the audience knows it (or something) is coming.
4. *Staging* – The idea of presenting an action so that it is unmistakably clear and is not missed by the audience.
5. *Follow through and overlapping action* – Guiding the termination of an action and establishing its relationship to the next action. Actions should flow into one another to make the entire scene flow together.
6. *Straight ahead versus pose-to-pose action* – Two contrasting approaches to the creation of movement. *Straight ahead* refers to progressing from a starting point

and developing the motion as you go. *Pose-to-pose* refers to the approach of identifying key frames and then interpolating intermediate frames between them.

7. *Slow in and out* – The spacing of the in-between frames to achieve subtlety of timing and movement. This is based on the observation that characters usually ease into and ease out of actions.
8. *Arcs* – Since things in nature don't usually move in straight lines, this helps in defining the visual path of action for natural movement.
9. *Exaggeration* – Accentuating the essence of an idea via design and action.
10. *Secondary action* – The action of an object resulting from another action. These support the main action, possibly supplying physically based reactions to the previous action.
11. *Appeal* – Creating a design or an action that the audience enjoys watching.

These principles were adapted for computer animation by Lasseter [86]. Squash and stretch, timing, slow in and out, arcs, and secondary actions deal with how the *physics* of the character (like its weight, size, and speed) is presented in relation to its environment. Exaggeration, appeal, follow through, and overlapping action are the principles that address the design of an action sequence. Straight ahead and pose-to-pose are concerned with contrasting production techniques for animation. Anticipation and staging define how an action is presented to the audience,

Animators developed the animation pipeline based on these principles. An animation develops as an amalgamation of ideas: the story, the characters, the continuity, and the relationships between scenes. The animation pipeline is a sequence of several steps that converts a story to a final animation.

1.2 The Animation Pipeline

First, a preliminary storyline is decided upon along with a script. Next, a *storyboard* that lays out the action scenes by sketching representative frames is developed. The story sketch shows character, attitude, expressions, type of action, as well as the sequence of events. In a preliminary storyboard, however, only the sequence of actions of the various characters are planned. The characters are not fully developed. The look and feel of a character is developed by sketching the character in various poses in a *model sheet*. The appearance of the character is documented from all directions and is used as a reference while actually animating the character. The *exposure sheet* records information for each frame such as sound track cues, camera moves, and compositing elements. Often the storyboard is transferred to film with the accompanying sound track and a story reel or an *animatic* is created to get a feel of the visual dynamics of the animation. Once the storyboard is fixed, a *detailed story* is worked out. *Key frames* (also known as *extremes*) are then identified and produced by master animators. Assistant animators are responsible for producing the frames between the keys; this is called *in-betweening*. *Test shots*, short sequences rendered in full colour, are used to test the rendering. The penciled frames are transferred to cels and painted. These cels are then composited together and filmed to get the final animation.

Computer animation production has borrowed most of the ideas from the conventional animation pipeline. The storyboard still holds the same functional place in the animation process, as does the model sheet. However, after the planning phase, computer animation often makes the transition into 3D. The character models and the world which they inhabit have to be handcrafted. Controls are provided that allow movements of various parts of the character. Then the animation staging is done in 3D, in which the camera positioning and movement for each shot is decided. This is followed by shading and lighting the animation and finally rendering the frames. As mentioned earlier, this sequence of steps is extremely tedious and time-consuming. It involves a lot of skill and a trial-and-error, iterative process wherein performing one task may require redoing one or more previously completed tasks.

We are now ready to examine moving-camera character animations and the challenges the animator has to face while creating them.

1.3 Moving-Camera Character Animation

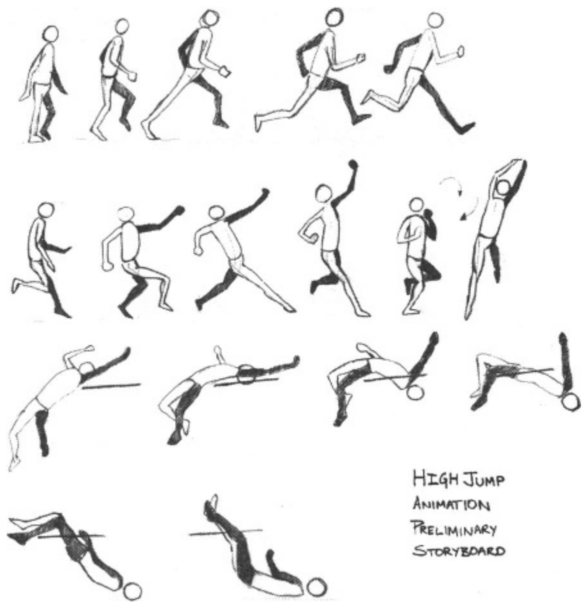


Fig. 1.1. A preliminary storyboard (see top to bottom, left to right).

We explain the creation of a moving-camera character animation using an example, *Hugo's High Jump*. Hugo [17] is the name of the character in the animation. We start with a preliminary storyboard (see Fig. 1.1) for this animation. Once the basic action has been planned and the character's look has been decided upon, we get the final or detailed storyboard, as shown in Fig. 1.2.

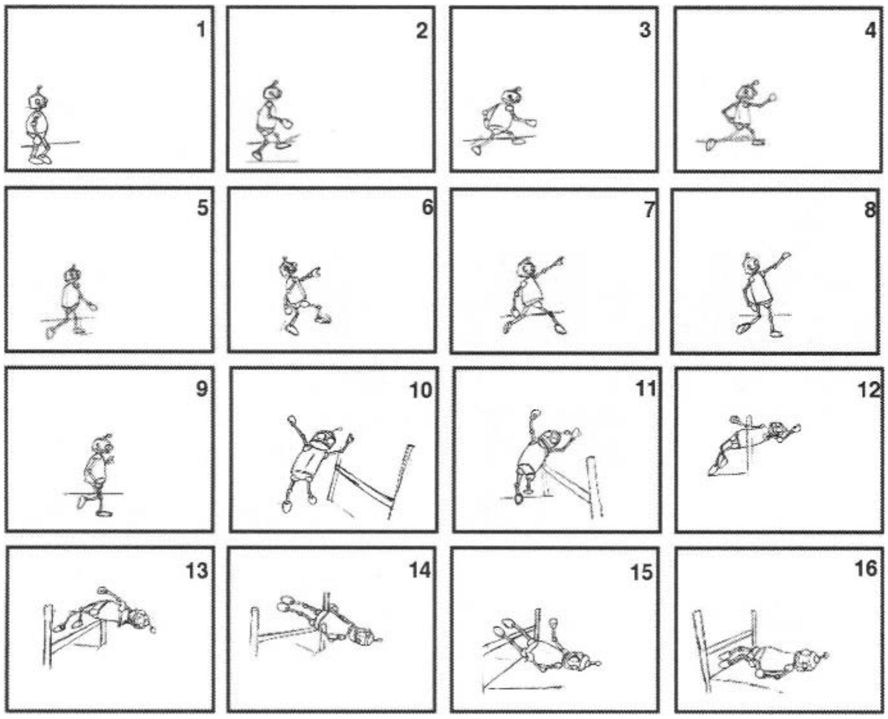


Fig. 1.2. The final storyboard for *Hugo's High Jump*.

Then the layout of the scenes is planned. Among other things, the layout also indicates the camera position for each frame. The layout is guided by the principle of staging and has to clearly portray where the viewer is supposed to be while observing the situation. Figure 1.3 is a time-lapse sketch for the animation sequence storyboarded in Figures 1.1 and 1.2. A time-lapse sketch shows the position of the character at different times in a single sketch.

This animation has a moving camera, i.e., the viewpoint is changing in each frame. The layout helps plan these camera moves. Figure 1.4 shows how the framing of the shots change as the camera moves through frames 1, 9, 11, and 15. When the character is seen from the camera position for that particular frame number, the scene looks like the corresponding thumbnail on the storyboard. The movement of the camera center is drawn in red across all the frames in Fig. 1.5. It is clearly seen that in order to achieve clarity in staging the animator has to create a very definite relation between the pose of the character and the camera position. Every shot is

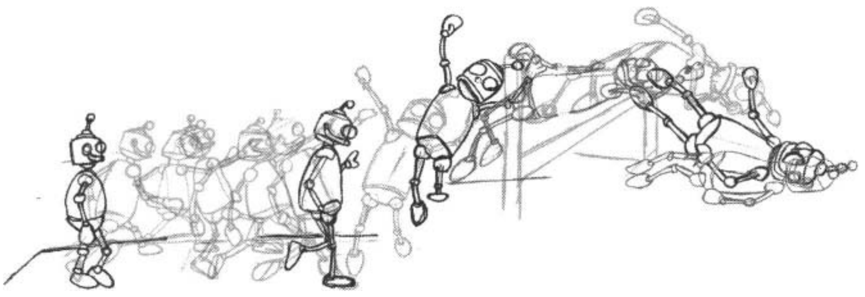


Fig. 1.3. A time-lapse sketch of *Hugo's High Jump*.

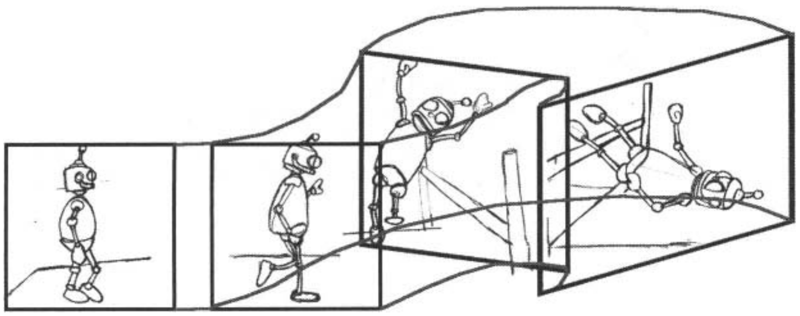


Fig. 1.4. The moving-camera frame (see colour insert).

drawn from the viewpoint of the audience, implicitly establishing a camera from which the action is understood clearly.

Thus, the camera–character relationship plays a pivotal role from a very early stage in the animation creation process. This combination of the camera or the *view* and the character's pose or movement is maintained throughout the creation of the animation, even when the character is transferred from two dimensions to three dimensions.

Translating the planned camera and character moves to 3D is an extremely difficult task. In this book we develop a framework to alleviate this problem. In order to illustrate the primary difficulty in creating moving-camera character animations, we evaluate the challenges involved in the process and suggest our alternative methodology.

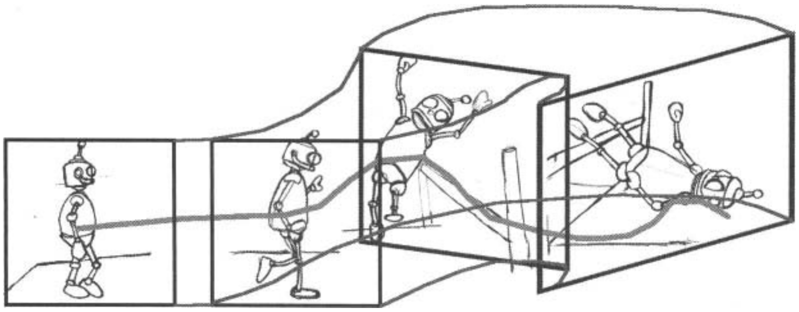


Fig. 1.5. Path of the camera center across all frames (see colour insert).

1.3.1 Challenges in creating moving-camera character animations

The animation has to be created in three dimensions. A mesh model of the character is made using the character's model sheets for reference. Then the character's posture, in three dimensions, has to be visually matched to the sketches for every key frame. This may require manually deforming various parts of the character. Moreover, the view direction from which the character is seen needs to match with the viewpoint in the sketch. This requires considerable effort on part of the animator because the possible combinations of a camera and a character's pose are often overwhelming. Next, to generate the animation, the character poses and the cameras for the in-between frames are obtained. In order to do this, if they are independently interpolated, then there is no guarantee that the in-between character pose will have its corresponding viewing camera as intended by the animator (see Fig. 1.6). A number of iterations may be needed to get the appropriate match between all the character poses and the cameras required to generate the desired animation.

A straightforward solution to this problem is to make the interpolation of the cameras depend on the interpolation of the poses in some manner, in order to preserve the association between them. This can be achieved rather simply, by making one interpolation function dependent on the other. Naive approaches, however, fail to present any geometric representation for this dependence. Thus, they do not provide sufficient insight into the structure of such a dependence and cannot be used to exploit the camera and character pose association for versatile animation authoring. They may also have to be modified on a case-by-case basis. In an alternative approach, which is more intuitive and requires significantly less work, we find the camera and the character's posture, which together best match the sketch, using computer vision techniques. In this way the camera and the corresponding character pose get associated when they are recovered together from the sketch. All the recovered cameras and their corresponding poses, taken together, form a space. This space provides a representation for all the moving-camera character animations that can be generated using the recovered cameras and poses. In particular, it explicitly embodies the association between the two. Now, in order to generate the in-between frames for the

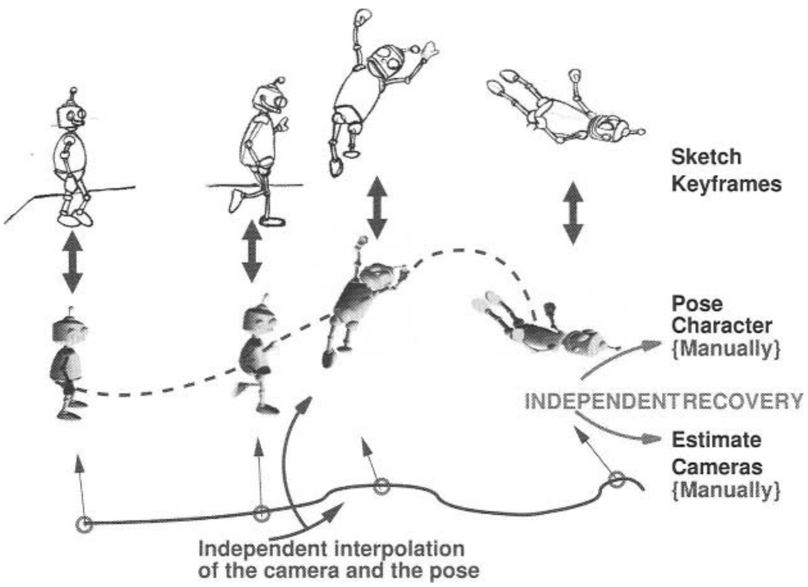

Sketch Keyframes
Pose Character {Manually}
INDEPENDENTRECOVERY
Estimate Cameras {Manually}
Independent interpolation of the camera and the pose

Fig. 1.6. Approach 1: Separately reconstructing poses and cameras from sketches and then independently interpolating both to get the desired animation.

animation we only have to interpolate the camera in this space (see Fig. 1.7). This interpolation automatically generates the corresponding character poses associated with the interpolated cameras. Moreover, the camera and character pose association is maintained throughout the animation.

This approach of associating the camera and the character pose is fundamental to *view-dependent animation*. This motivates us to investigate the concept of view-dependent character animation for creating moving-camera character animations. View-dependent animation explores the complex relationship between the camera and the character's actions and, hence, solves the staging problem in a limited sense. In view-dependent animation, the character's action *depends* on the view. An animator specifies the character pose with the desired view direction for certain *key views*. This is done on the basis of prior planning by the animator in the form of storyboards or key frame sketches. Then our framework generates a space using the key cameras and associated character poses specified by the animator. In order to generate the desired animation, the animator has to trace the planned camera path in this space. The corresponding sequence of character poses is generated automatically in response to the camera movement. It is also possible to quickly examine this space and try out other variations of the camera path, which can generate other interesting animations.

The principle of staging dictates that the character's action is to the camera so that the intent of the action is clear and not obscured. In general, the character's action is considered independent of the camera used to render the animation. View-dependent animation gives us a different perspective to the problem of camera–character asso-

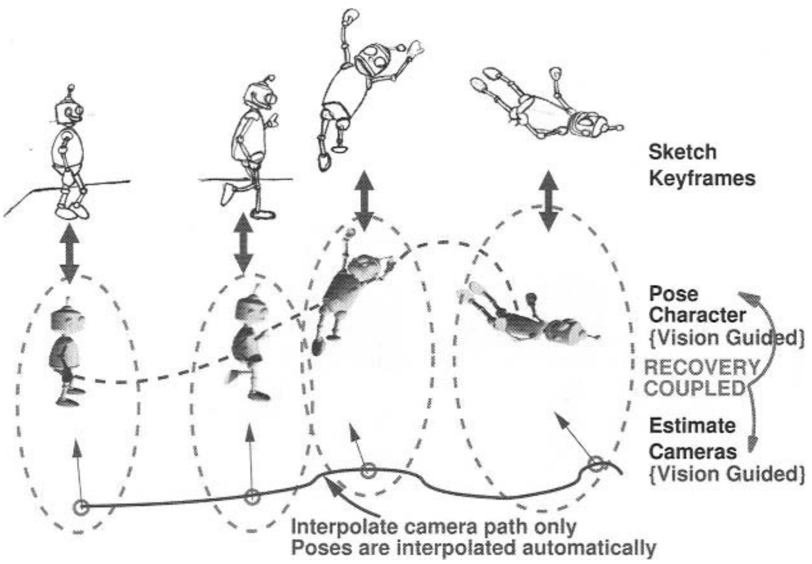

Sketch Keyframes
Pose Character {Vision Guided}
RECOVERY COUPLED
Estimate Cameras {Vision Guided}
Interpolate camera path only
Poses are interpolated automatically

Fig. 1.7. Approach 2 (view-dependent approach): Computer vision-based techniques allow coupled camera and pose recovery, and then interpolating only the camera generates the desired animation.

ciation. Our framework captures the relationship between the camera and the character pose based on the animator's specification of key views. It generates a space that characterizes the camera–character pose relationship desired by the animator, based on inputs from the animator. Once this space is created, the framework automatically maintains the camera–character pose correspondence. This allows the animator to concentrate solely on the aesthetic component of the desired animation. It, thus, translates the animator's intuition and her concept of staging and layout of moving-camera character animations into a tangible, explorable space of cameras and character poses.

The view-dependent approach demands that we define a formal representation of the camera–character pose association. This representation should be practical to implement and use. It should obviously be conducive to the many ways in which a camera can be defined and interpolated. It should also encompass the character pose variations. These requirements span over a multitude of challenging computer graphics and computer vision issues. In the subsequent sections, we examine the scope of and the challenges we face in designing a framework for view-dependent animation, and how we solve them.

1.4 Designing a Framework for View-Dependent Animation

We build a framework that facilitates the creation of view-dependent character animations, i.e., the animation must respond automatically to changes in the rendering viewpoint. In order to design a general framework that encapsulates the rich diversity offered by moving-camera animations, we are faced with a number of challenging problems, as described below:

1. The framework must provide a sound basis for representing view-dependent animations.
   - Since a view-dependent animation is a combination of views and associated character poses, the framework must have a way of encapsulating information about viewpoints and their associated character poses.
   - It must also be able to generate new animations quickly, with minimal effort, after the initial set of views and poses have been specified.
   - It should be possible for an animator to add views and poses to an existing set, in order to enhance the animation. The framework must be able to handle such augmentations.
2. The framework should be able to represent and exploit all the variations possible in defining a camera shot. Camera shots, in an animation, vary from wide pans to close ups, often as guided by cinematographic or theatrical principles [6]. The variations we want to capture are
   - Changes in the view direction
   - Changes in the distance of the camera center (or viewpoint) from the character
   - Changes in focal length of the camera, i.e., zoom and scaling
3. The animator has to specify the desired animation using some mode of input to the framework. Traditionally, animators are familiar and trained to work with sketches. We have already seen an example of how sketches are used to plan an animation. The framework must have the following capabilities when dealing with sketches:
   - It must be able to recover information about the intended viewpoint from the sketch.
   - It must be able to assist the animator in posing the character using the sketched pose as a guide.
   - The framework should work with a large variety of character sketches. Sketches are not photo accurate. They are often rough representations of the character.
4. Animators often use recorded video performances as references for keyframing characters (i.e., specifying an animation using a sequence of key frames). Numerous examples [115, 124], of the use of this technique for keyframing humans and animals can be found. The framework should be able to use video input to create new view-dependent animations.
   - The framework must have a way to describe and interpret the information contained in a video in terms of the cameras in each frame and the corresponding pose of the character.
5. It may be desirable to mix many modalities while creating the animation such as keyframing, animation from reference video, and motion capture [115]. Suppose

the animator wants to replicate the camera movement of a master cinematographer from some existing movie (video) in the animation. She, however, wants to give a unique movement style to the character and, hence, wants to keyframe the movement separately, using sketches to plan the poses. Currently, this mix and match would require tremendous manual effort. We want our framework to adapt to such multimodal inputs, i.e., it should work even when the input method is a combination of these modes.

6. One of the primary objectives of the framework is to aid the animator. In order to achieve this, the framework must adhere to the following principles:
   - It must, above all other considerations, allow sufficient control and freedom to the animator so that the desired animation can be generated.
   - The framework is not meant to replace the animation pipeline. Rather it is supposed to complement existing animation work flows by expediting the creation of complex moving-camera character animations. For this purpose, the generated view-dependent animation must blend in seamlessly with animations generated using more conventional techniques, like keyframing.

This book presents a framework meeting all the challenges enumerated above. We introduce the concept of a *view space* defined by the key views and associated key character poses that completely captures all the information required to produce a view-dependent animation. The framework generates new animations in real time whenever the animator traces out a new camera path on the view space. We show that simple interpolation schemes allow the generation of in-between poses from key poses by just defining the intermediate camera positions and orientations.

We present complete pipelines to create view-dependent animation from inputs sketches, videos, and a mix of both. The ability to map sketches and video into a common representation (i.e., the view space) allows us to mix and match these various input types to create unique view-dependent animations. The technique allows the mixing of other input modalities such as motion capture data as well.

The framework allows the animator to seamlessly blend view-dependent animations with non view-dependent animations[1] by simply matching the rendering cameras for successive frames.

Finally, we look at a very interesting application of view-dependent animation to reuse stylized animation. We present a formulation to synthesize an animation by reusing the view-dependent instances of a single character and a group of characters. We also show how one can animate different parts of a character using different view-dependent variations.

1.5 Tour of the Book

Animations where the character and the rendering camera both move are known as moving-camera character animations. The sheer number of parameters the anima-

[1] Non view-dependent animations are those in which the character's pose does not explicitly depend on the camera. See Section 3.6.3 for more explanation.

tor has to control, in order to get the desired action shot from the intended camera position, is overwhelming. In this book we present view-dependent animation, as a solution to the challenges encountered during the creation of moving-camera character animations.

This is our primary motivation for developing a framework for view-dependent animation. A quick overview of the various topics discussed in this book are as follows:

- Chapter 2 presents the theoretical description of our framework. It presents the concept of a view space and how it encapsulates the camera–character pose relationship. Further, this chapter explains how the view space represents and uses all variations possible in defining the camera parameters. It also presents an example to illustrate these concepts.
- All animators, regardless of whether they are creating 2D or 3D animations, start from model sheets and storyboards. Sketches are perhaps the most common and familiar medium of input among animators. Chapter 3 presents our pipeline for creating view-dependent animations from sketches. It introduces two novel algorithms for view-dependent posing and view-dependent mesh deformation. These allow the animator to create view-dependent models from sketches more intuitively and efficiently.
- In Chapter 4 we present our technique for creating view-dependent animations from multimodal inputs. We first analyze the general problems in authoring view-dependent animations from multimodal inputs. We then present a solution to these problems by demonstrating how our framework can use video-based input to generate view-dependent animations. We argue that this framework can handle multiple types of inputs and that they all share a common representation in terms of the view space. This allows the animator to mix and match these inputs as desired. We also present examples to demonstrate the use of multiple input modes in creating new and interesting animations.
- The view-dependent animations generated by a camera path is unique. In Chapter 5 we develop a framework for reusing the view-dependent variations in order to synthesize novel animations. We present three techniques for reusing camera-controlled pose variations to animate multiple view-dependent instances of the same character, a group of distinct characters, or the body parts of the same character. We present animation examples for illustrating each of these techniques.
- Chapter 6 presents a concise summary of the features of our framework for view-dependent character animation. We conclude by presenting some directions for future work as a set of interesting problems, which can be solved by extending the ideas presented in this book.

A Framework for View-Dependent Animation

In the previous chapter we have seen that an animation is generated as a consequence of some action captured from a desired camera. In a moving-camera character animation, the character's pose depicting an action or motion needs to be defined in tandem with the camera. We provide a framework that embodies the concept of camera and character pose association.

2.1 Prior Work

The essential idea behind the framework for view-dependent animation is that it provides a formal representation of the camera–character pose association. There have been attempts toward developing various representations of the camera as well as the character pose. However, these representations do not capture the association between the two. In this section, we present a brief discussion of these representations, in light of the objective we are trying to achieve.

We first look at works that represent plausible character poses as an abstract space but control the character pose animation by some mechanism other than the camera.

2.1.1 Character pose spaces for animation

Lewis et al. [90] present a method to perform *pose space deformation*. Here, deformation is represented as a mapping from a pose space, defined by either an underlying skeleton or a more abstract system of parameters, to displacements in the object's local coordinate frames. They use scattered data interpolation in the pose space to generate intermediate poses for the animation, using a radial basis function. This technique is suitable for shape interpolation and layered skeletal animation; however, it has no way of associating the pose space with the view.

Ngo et al. [104] present a technique that models the space of all the key configurations (called *key poses* in this book) as a cross product of simplicial complexes. Then they define the mapping from this space to the image space and show that this mapping is invertible. This allows the user to manipulate the image without understanding the structure of the configuration-space model. Their system applies simplicial configuration modeling to 2D vector graphics. This idea is similar to the convex hull structure used in [109] (see Section 2.1.3), which is also a simplicial complex.

In another recent work Igarashi et al. [67] present the technique of *spatial keyframing*. The key poses are defined at specific positions in a 3D space. The mapping from the 3D space, to the configuration space is defined by an interpolation function. The user controls a character by adjusting the position of a control cursor in the 3D space and the pose of the character is given as a blend of nearby key poses. Thus, the user can create motion in real time that can then be recorded and interpreted as an animation sequence. Spatial keyframing associates key poses of the character with directions in 3D space. However, it does not associate the character pose with the view direction.

Next, we examine works that create interesting animations using various representations of the rendering camera. These methods vary the camera parameters, independent of the character pose, in order to generate the animation.

2.1.2 Controlling camera variations to create animation

Agrawala et al. [4] present a multiprojection rendering algorithm for creating multiprojection images and animations. They develop an interactive interface for attaching local cameras to the scene geometry to alter the projection for each object independently, thereby generating a multiprojection image. Singh [119] also presents a technique for constructing a nonlinear projection as a combination of multiple linear perspectives. The viewports of a number of exploratory linear perspective cameras are laid out on a common canvas on which the nonlinear projection of the scene is rendered. Each exploratory camera influences different regions in the scene based on local weight values. This approach neither integrates well into a conventional animation work flow nor has ways to control global scene coherence.

Coleman and Singh [29] make one of Singh's [119] exploratory cameras a *boss* (or primary) camera; this camera represents the default linear perspective view used in the animation. All other exploratory (or secondary) cameras, when activated, deform objects such that when viewed from the primary camera, the objects will appear nonlinearly projected. They describe a framework for the interactive authoring of nonlinear projections, defined as a combination of scene constraints and a number of linear perspective cameras. These techniques used in the production of the short animation movie *Ryan* [85] demonstrate how geometric and rendering effects resulting from nonlinear projections can be seamlessly introduced into current production pipelines. This type of camera-based stylization can produce striking effects, which can be aesthetically harnessed by an artist to create interesting animations.

In contrast to the above techniques, Yang et al. [142] extend traditional 2D image deformation techniques to 3D space and perform the deformation only on the 2D frames generated by the graphics pipeline. This requires no change in the traditional graphics pipeline. They derive the deformation algorithms from 3D nonlinear perspective projections, which consider factors such as depth, view angle, and camera position.

Other work involving camera or view direction control has chiefly focused on permitting a user to manipulate a virtual camera in a virtual environment as presented in [45, 46, 54]. Funge et al. [42] construct a cognitive model, which embodies the knowledge of the director and the cinematographer controlling the camera, using a cognitive modeling language. The camera acts like a cognitive agent and places itself based on the cues generated from the application rendering the scene and the axioms defined to govern its behaviour in the cognitive model.

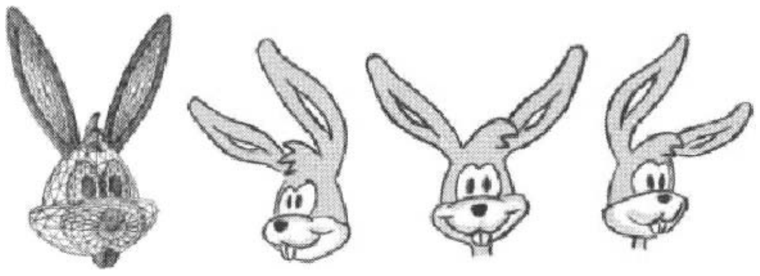
Although it is evident from the above discussion that many different representations of the character pose spaces and rendering cameras have been investigated in the past, there is very limited prior work on the idea of view-dependent animation, i.e., techniques that actually use the camera to influence the pose of the character. We now discuss techniques from the existing literature, which use the idea of view-dependence of the character pose.

2.1.3 View-dependent geometry

The idea of dependence of the character's geometry on the view direction was first put forward by Rademacher [109] in his work on *view-dependent geometry* (VDG). He draws inspiration from the fact that artists catalogue the appearance of a character on a model sheet. Since these are hand-created images, they do not correspond to a precise physical space. They are drawn to achieve the best aesthetic effect and are not bound to geometric precision. As a result, these drawings typically contain many subtle artistic distortions, such as changes in scale and perspective, or more noticeable effects such as changes in the shape or location of features. VDG allows the animator to specify models in such a manner that their *geometry can change with the viewpoint*, hence capturing different looks of an object from different viewing directions.

The inputs to the system are a 3D model of a character (the base model) and a set of drawings of the character from various viewpoints [see Fig. 2.1(a) and (b)]. First, the user manually aligns the base model with each drawing by rotating and translating the camera. This gives a best matching viewing direction for each sketch, which is known as a *key viewpoint*. The user then manually deforms the aligned base model by altering the positions of the vertices of the mesh model, in order to match it with the drawing. Note that the topology (vertex connectivity) of the model does not change during the deformation; only the vertex locations are altered. Also note that the drawings are not altered; only the base model is deformed. The deformed mesh model is called a *key deformation*. The process is shown in Fig. 2.2.

A key viewpoint and a key deformation pair together constitute a *view-dependent model*. Such a view-dependent model is obtained for each sketch. These view-dependent models are constructed *a priori* in the modeling phase. In order to generate the animation, we need to determine the pose of the 3D model associated with any given camera direction.



(a) Base Model (b) Reference Sketches

Fig. 2.1. Inputs to the VDG system (images courtesy Rademacher [109]).

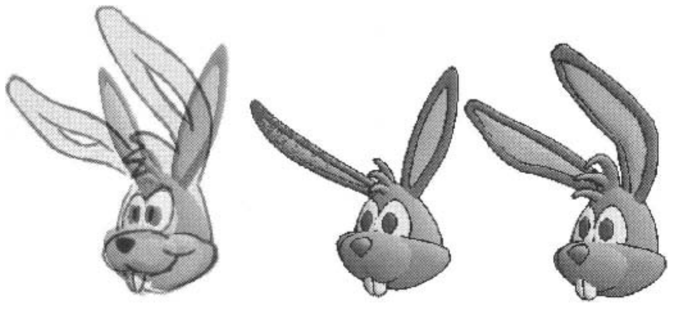


Fig. 2.2. Construction of the view-dependent model. First align the base model to the sketch and establish a key viewpoint. Then deform the model to match the sketch. On the right we see the final key deformation (images courtesy Rademacher [109]).

Given a view direction, the shape of the corresponding 3D model is determined as follows: The key viewpoints map to points on a sphere around the object, called the *view sphere*. This is so because this method considers only the viewing direction for the key and current viewpoints and not the distance from the cameras to the object. A convex hull of these points is constructed. At rendering time, the face of the convex hull, which is intersected by a ray from the current camera to the sphere center, is determined. The intersected triangle denotes the closest three key viewpoints surrounding the current camera. The current shape of the 3D model is generated as a barycentric blend of the key deformations associated with the closest three key viewpoints (see Fig. 2.3). Now, tracing any camera path on this view sphere generates the appropriate animation with view-dependent deformations.

This method also supports creation of animated view-dependent models. In this case, the base model is nonrigidly animated, and a single set of key deformations is not sufficient. Such situations need a different set of key deformations for the key frames of the model's animation. This essentially results in a separate view sphere

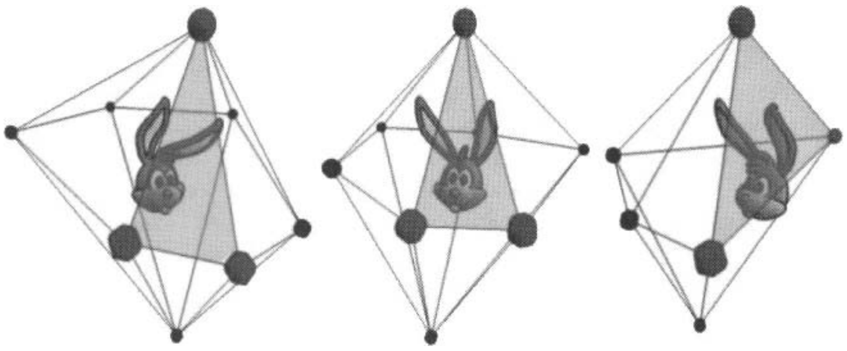


Fig. 2.3. Viewpoints for each key deformation are shown as spheres around the model. To compute the shape as seen from the current viewpoint, find the nearest three key viewpoints and blend the corresponding key deformations (images courtesy Rademacher [109]).

at each key frame. The animation is generated by blending the deformations on a per-frame basis, in response to the current viewpoint as the viewpoint moves from one view sphere to another.

From Phong shading [107] to view-dependent texture mapping [37], graphics research has shown that gaze direction is an important parameter in rendering objects. This work extends this progression by modifying the actual *shape* of an object depending on where it is viewed from. In doing so, they directly address a problem in 3D animation — the loss of view-specific distortions as an object moves from the artistic 2D world to the geometric 3D world. By employing view-dependent geometry in animation, we can render 3D models that are truer in shape to their original 2D counterparts.

2.1.4 Observer-dependent deformations in illustrations

Illustration has some visual characteristics that are very interesting although very difficult to obtain using a computer. While the simulation of various painting styles (see Section 5.1.1) has been successfully applied to computer-generated imagery, expressive capabilities have not been developed to the same extent. The deformations of objects and space is a major element in the expressiveness of illustration. Martín et al. [99] use hierarchical extended nonlinear transformations (HENLT) to produce *observer-dependent deformations* in illustrations, in order to capture its expressive capabilities. The HENLTs change with variation in the observer's position and orientation. They are then used to deform the object. So the object is seen differently from different directions.

The techniques of Rademacher [109] and Martín et al. [99] are the only known direct applications of the view-dependent technique to animation, other than the one presented in this book.

In all the representations, except [99] and [109], of the camera and the character pose discussed above, there is no direct one-to-one correspondence between the viewpoint and the pose of the character. Hence, they are not particularly suited to address the problem of moving-camera character animations in general. We present a framework that embodies a general representation of view-dependent character animation. We use a *view space* (see Section 2.2), which is a space over camera parameters. We associate a pose with every view direction in the view space, thus in essence creating an auxiliary character pose space. The framework allows the use of general forms of configuration-space models, as well as a simplicial complex, to represent the view space. We also use a radial basis interpolant to blend the selected key poses to generate the pose associated with the viewpoint in question (see Section 3.6.2).

We show that the framework we present reduces to the VDG formulation as a special case (see Section 2.2). In addition, we also present techniques for automated recovery of cameras and creation of view-dependent models from sketches, videos, and hybrid inputs. In doing so we reduce the amount of manual intervention required in the creation of the view-dependent models, thus alleviating one the major limitations of the VDG technique. Kate et al. [76] propose a method to automate some aspects of the view-dependent model creation. We present interactive techniques that allow semi automated creation of the view-dependent models. These are not only intuitive but are also robust and offer the animator more control over the animation.

We now present the details of the framework for view-dependent animation. In the following section we examine how the view space is formed from key views and associated character poses.

2.2 The View Space

We assume, for simplicity of explanation, that we are animating a single character and that the camera is looking toward the character (i.e., the character is in the field of view of the camera). We also assume that the view direction is a unit vector.

At a given instant of time the character may be potentially viewed from a set of different viewpoints. The character may possibly have a different pose associated with each of these viewpoints (see Fig. 2.4). We consider such a set of viewpoints and associated character poses as one sample. We define a representation that enables aggregation of such samples as an ordered sequence. These sets of viewpoints and associated character poses sampled (or ordered) across time form a *view space* (see Fig. 2.5). We refer to this time (which orders the samples) as the the *sampling time* or *sampling order*. Every point on the surface *envelope* of this view space represents a viewpoint (and a unit view direction), v. If we do not consider the sampling order, then the view space is simply the space formed by the viewpoints and their associated character poses. Typically the animator provides only a finite number of samples to construct the view space. However, the resulting space is a continuous space. Since for every viewpoint there is a unique view direction, we use these terms interchangeably. We denote the pose of the character, associated with a view direction v, as m_v. A *character pose*, in this book, is the resulting mesh model of the character having

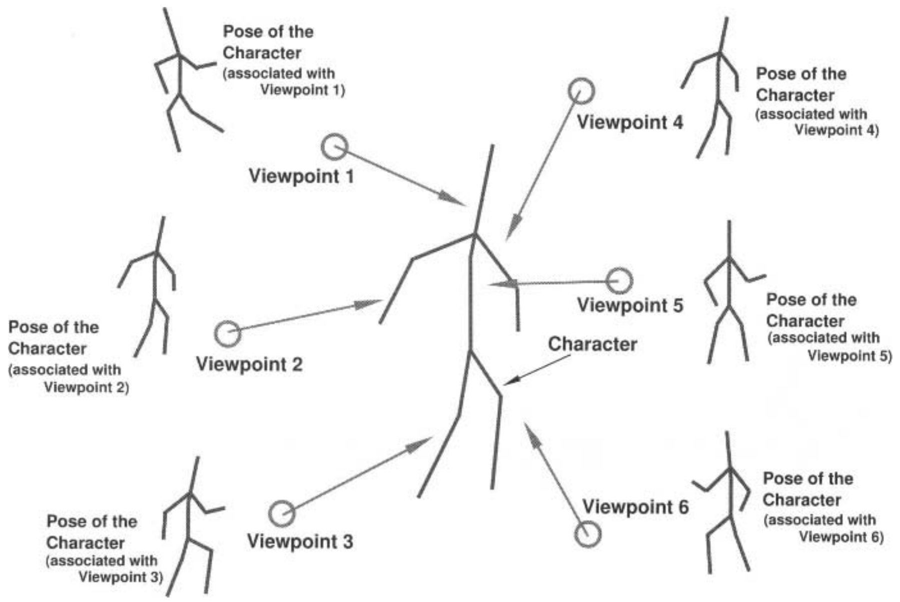

Pose of the Character (associated with Viewpoint 1)
Viewpoint 1
Viewpoint 4
Pose of the Character (associated with Viewpoint 4)
Pose of the Character (associated with Viewpoint 2)
Viewpoint 2
Viewpoint 5
Character
Pose of the Character (associated with Viewpoint 5)
Pose of the Character (associated with Viewpoint 3)
Viewpoint 3
Viewpoint 6
Pose of the Character (associated with Viewpoint 6)

Fig. 2.4. A character may be potentially viewed from a set of different viewpoints at a given instant of time. A different character pose may be associated with each viewpoint.

undergone any change that may be rigid or nonrigid, i.e., it includes mesh deformations as well as changes in the mesh due to articulation of the embedded skeleton (see Section 3.3). We couple the character pose to the view direction. Hence, changing the view direction changes the pose of the character.

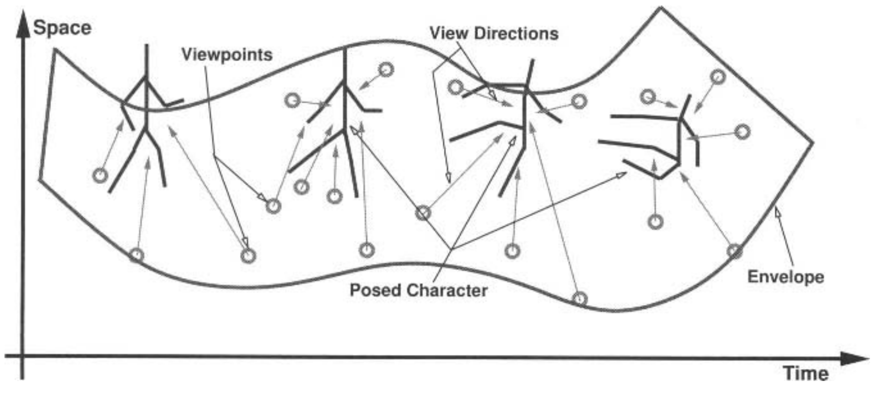


Fig. 2.5. A view space as an aggregation of all the sets of viewpoints. One character pose is shown for each set of viewpoints.

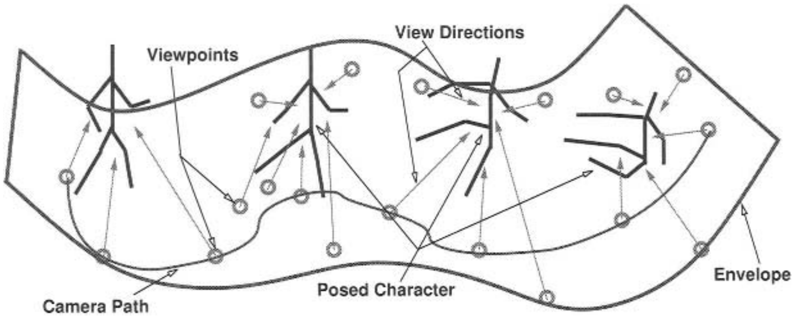


Fig. 2.6. Tracing a camera path on the envelope of the view space generates an animation.

An animation is generated by tracing a path, P, on the envelope (see Fig. 2.6). A point p on this path consists of the view direction associated with the point on the envelope, $\underline{v}$, and is indexed by time (run time of the animation) along that camera path, $\underline{t}$, measured from the start of the camera path. Note that the run time of the animation should not be confused with the sampling time. We refer to points on a camera path P, as $p \equiv (\underline{v}, \underline{t})$. The animation generated is the sequence of the poses $m_{\underline{v}}$ associated to $\underline{v}$ on the path P viewed along the direction $\underline{v}$. Every distinct camera path generates a distinct animation. This is the basic idea behind the framework.

In order to create the view space, the animator provides a set of *key viewpoints* or *key view directions* and the associated *key poses*. Let v^k represent a key viewpoint and m_{v^k} represent the associated key character pose. The animator can provide these in the form of a set of sketches, a video, or a mix of the two. In the *Hugo's High Jump* animation (first discussed in Chapter 1), the animator provides the sketches for the key frames (see Fig. 1.2). We use our framework to extract the key view directions and key poses from these sketches (we describe the process in Chapter 3). These form the *view space* on which the animation is generated. Figure 2.7 shows the sketches provided by the animator and the corresponding key views created using the framework. In the bottom row, we show the key poses as seen from the key view directions. Figure 2.8 shows the recovered camera centers (viewpoints) and view directions, shown from an independent camera.

Note that for each view, the sphere centered at the look-at point (in this case the end of the unit length view direction vector) is the set of all possible view directions from which one can look toward that point. Hence, this sphere may be thought of as a view space generated by just one view. The complete view space is, therefore, the union of the view spaces generated by all the views (see Fig. 2.9).

In order to generate an animation along a camera path, $P(\underline{v}, \underline{t})$, on the envelope of the view space, we need to generate the associated character pose, $m_{\underline{v}}$, for every point p on P. To do this, for any view direction $\underline{v}$, we determine the r-closest key viewpoints (closest in the metric defined on the envelope), v_j^k. An example of such a

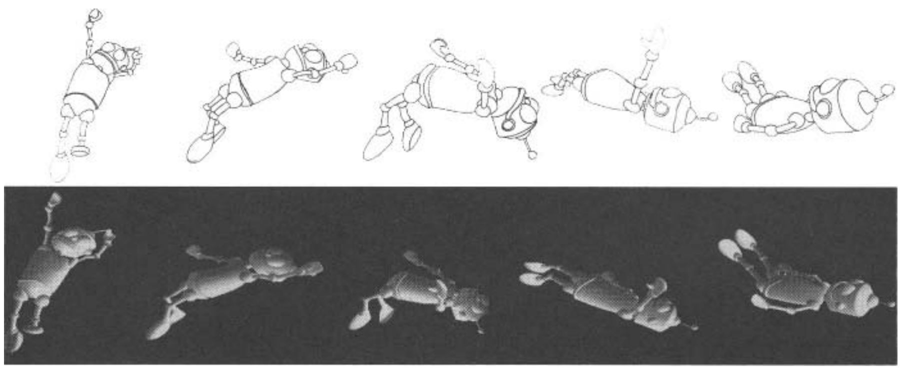


Fig. 2.7. The top row shows the sketched poses given by the animator. The bottom row shows the reconstructed key views.

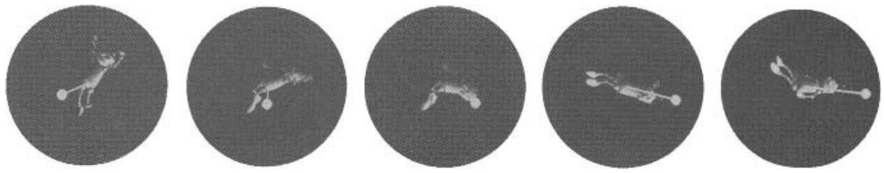


Fig. 2.8. The small sphere is the recovered camera position, and the line shows the view direction vector. The larger sphere, centered at the look-at point, gives an idea of the relative positioning of the recovered camera centers (see colour insert).

metric may be the geodesic distance between the viewpoints measured on the surface envelope.

For clarity, henceforth we represent v^k as v and v^k_j as $\bar{v}$. The character pose $m_{\underline{v}}$ is then given by

$$m_{\underline{v}} = \sum_{\bar{v}} w_{\bar{v}} m_{\bar{v}} \, . \tag{2.1}$$

Thus, $m_{\underline{v}}$ is a weighted blend of the corresponding $m_{\bar{v}}$'s (i.e., the r-closest key view poses). The $w_{\bar{v}}$'s are the corresponding blending weights. The $w_{\bar{v}}$'s vary inversely to the proximity of $\bar{v}$ to $\underline{v}$ [see Equation (3.14) in Section 3.6].

An example of a path, $P(\underline{v}, \underline{t})$, is shown in Fig. 2.9. Figure 2.10 shows the selection of the r-closest key viewpoint for a given position of the rendering camera on the path.

The path shown in Fig. 2.9 is obtained by smoothly joining the key viewpoints. Some frames from the animation obtained from this path are shown in Fig. 2.11. Here we see that the generated animation matches the planned storyboard frames very closely and the path generates the animation originally intended by the animator. This complete process is very intuitive for the animator as she does not have to worry about the camera and the character separately, once the view space has been

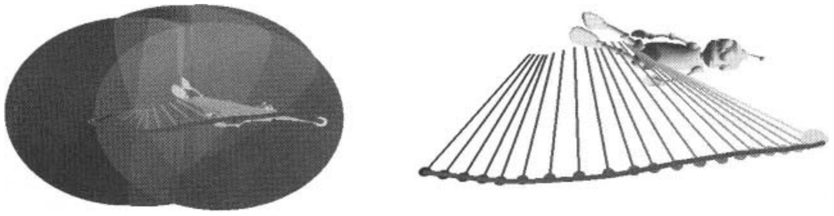


Fig. 2.9. The left image shows the path traced on the envelope of the view space. The right image shows a close-up view of the path. The larger green sphere at the end of the path shows the position of the (current) camera when this snapshot was captured (see colour insert).

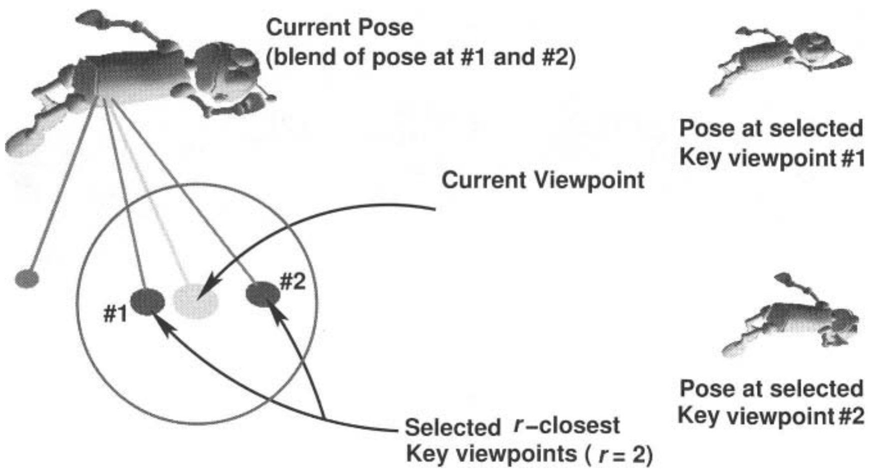


Fig. 2.10. The r-closest key viewpoints selected for a given position of the current viewpoint, and the corresponding character pose generated as a blend of the selected key poses.

created. We show later that other paths on this view space also produce interesting animations. Equation (2.1) computes $m_{\underline{v}}$ as a linear blend of the r-closest key view poses. Note that in calculating $m_{\underline{v}}$, the topology (vertex connectivity) of the model does not change; only the vertex locations are altered as every vertex in $m_{\underline{v}}$ is a weighted blend of the corresponding vertices in key view poses. This does not, however, guarantee an in-between pose, at an interpolated viewpoint on the camera path, in which the mesh will not self-intersect. This happens when the key poses being interpolated are very different from each other. This is a common problem with all interpolation-based animation techniques. The solution to this problem, in our case, is to add another key pose at that interpolated viewpoint such that the key pose matches the correct or desired in-between pose. We assume coherence over a local neighbourhood around any viewpoint, both in terms of the view direction as well as the character pose, i.e., the pose specified by the animator for any viewpoint is similar to the pose specified for any other viewpoint in its small neighbourhood. This guarantees spatio temporal continuity in the generated animation, i.e., the animation will not have any sudden *unwanted* changes in the view or pose between successive frames.

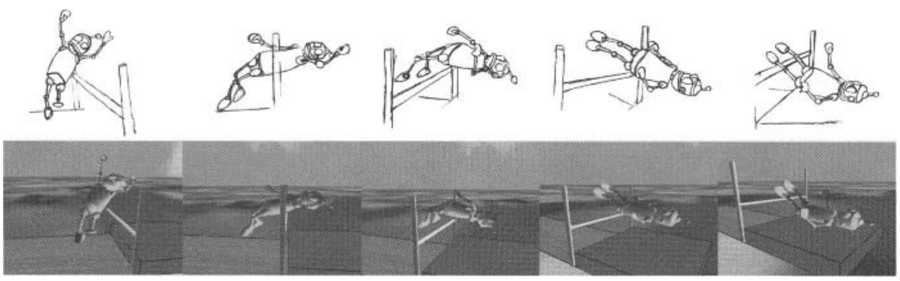


Fig. 2.11. The top row shows the planned storyboard. The bottom row shows the final rendered frames of the animation generated by the path shown in Fig. 2.9.

We assume that we are looking toward the character. This does not mean that all the view directions are directed toward a particular point on the character's mesh model. It only means that the character is in the field of view of the camera.

The view space for this example (shown in Fig. 2.9) is an instance of the general view space formulation. The view space can have other forms depending on the spatial location and sampling order of the sets of viewpoints used to construct it. The conditions under which they are generated are enumerated below:

1. If all the view directions, corresponding to a set of viewpoints sampled at a given instant of time, intersect at a common point (i.e., they share a common look-at point), then the instantaneous view space is a single sphere (also called a *view sphere*) centered at the point of intersection. This is trivially true if there is only one view direction for some time instant. If this condition holds for all sampling time instants, then the view space is an aggregation of view spheres. The spatial location and sampling order of these sets of viewpoints (i.e., view spheres) gives rise to the following view space configurations:
   a. If there is only one set of viewpoints (i.e., there is only one sample), then the view space is a single view sphere [see Fig. 2.12(a)].
   b. If there are multiple sets of viewpoints and each set is located at a different point in space and sampled at a different time instant, then the view space is an aggregation of view spheres separated in both space and time [see Fig. 2.12(b)]. The view space shown in Fig. 2.9 is an example of such a case (with only one view direction for each time instant).
   c. If there are multiple sets of viewpoints at the same spatial location, sampled at different time instants, then the view space is an aggregation of view spheres separated only in time and not in space [see Fig. 2.12(c)].
2. If all the view directions, corresponding to a set of viewpoints sampled at a given time instant, do not intersect at a common point, then the instantaneous

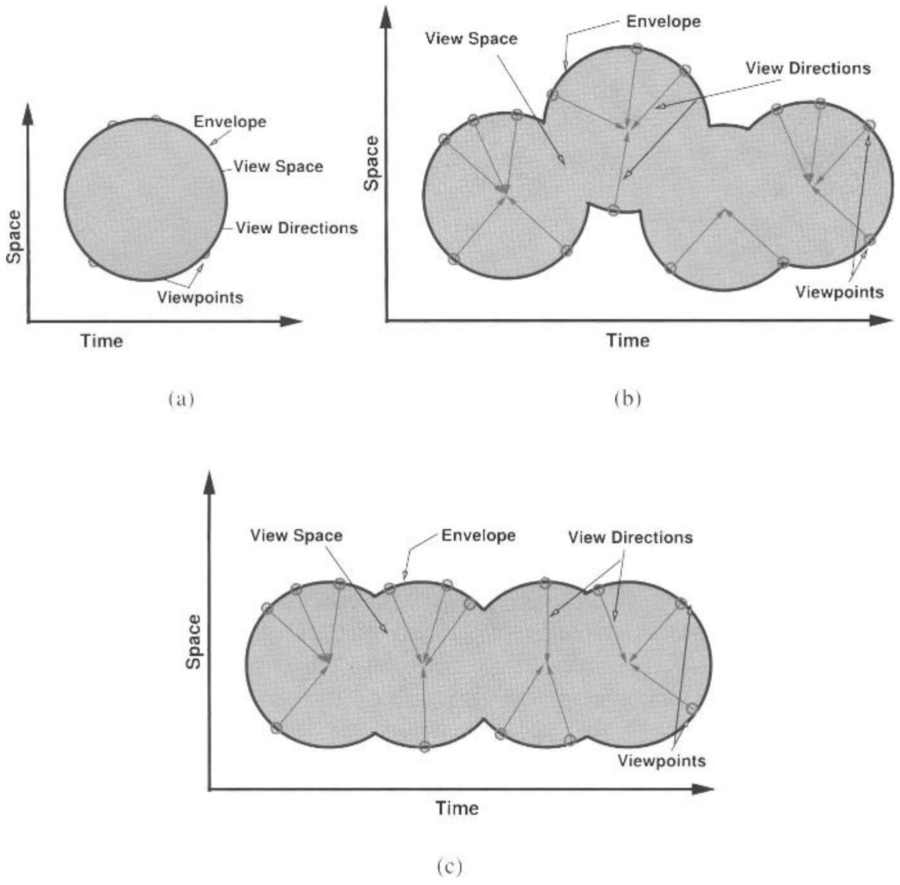


Fig. 2.12. Possible view space configurations: (**a**) only one set of viewpoints; (**b**) multiple sets of viewpoints and each set is located at a different point in space and sampled at a different time instant; (**c**) multiple sets of viewpoints at the same spatial location, sampled at different time instants.

view space is not a single sphere. It can be considered as a collection of spheres (one centered at each distinct look-at point). Then the complete view space is an aggregation of such instantaneous view spaces. The view space may have any of the three configurations analogous to the ones described above.

In the work by Rademacher [109] the view sphere formed by view-dependent models is a special case of our view space. Here, a convex hull of the viewpoints is computed. This partitions the view space by imposing a triangulation on it (see Fig. 2.2). A novel view-dependent model for any new viewpoint is generated by a barycentric blend of the key deformations at the vertices of the triangle in which the new viewpoint lies. This is clearly a special case of our novel view generation strategy on the envelope. Here, r = 3-closest key viewpoints set up a local barycentric basis for the novel viewpoint. The new character pose associated with this viewpoint is computed as a weighted blend of the key poses at the selected key viewpoints, using the barycentric coordinates of the novel viewpoint as weights. The major limitations of Rademacher's formulation are

- It does not handle the distance of the viewpoint, which is crucial for incorporating zoom effects.
- It cannot handle cases where all the camera view directions do not intersect at a single look-at point (the center of a view sphere), thereby limiting the method considerably.

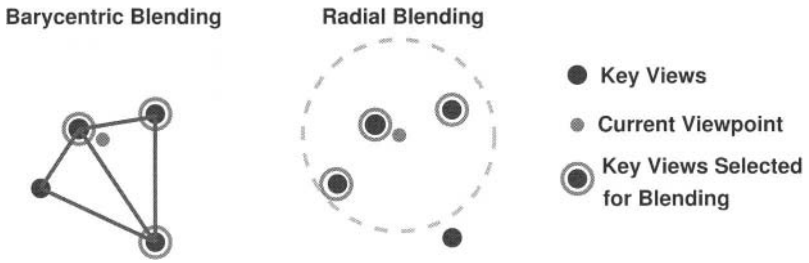


Fig. 2.13. Barycentric blending is biased toward choosing key viewpoints belonging to the same triangle. Radial blending does a better job in choosing the r-closest key viewpoints.

In following sections we provide ways to deal with both of the above. Further, the barycentric blending policy may also sometimes choose key poses that are farther away if they belong to the same triangle as the current viewpoint (see Fig. 2.13). The barycentric blending, however, has the advantage of being very easy to compute. Hence, the framework allows the user complete freedom in choosing the r-closest key viewpoints and blending the corresponding key poses. We can also use barycentric blending for this purpose, if required.

2.3 Distance of Viewpoint

In the previous discussion, we developed the framework considering only the view direction without the distance of the viewpoint. Now we add the component of distance to the framework, i.e., we want the character's pose to change as the distance of the viewpoint changes (with or without an accompanying change in view direction).

We assume that a tuple list (d_v^l, m_v^l) is associated with every view direction, v, forming the view space. Here, d_v is the distance of viewing and the associated character pose is m_v. The list is sorted on the distance field of each tuple. If the list has L elements, then $1 \leq l \leq L$. So the m_v^l's are the different poses of the character along a view direction at various distances d_v^l. As we change the distance, $d : d_v^{l1} \leq d < d_v^{l2}$, along a view direction, v, the resulting character pose is a blend of the character poses m_v^{l1} and m_v^{l2} (see Fig. 2.14).

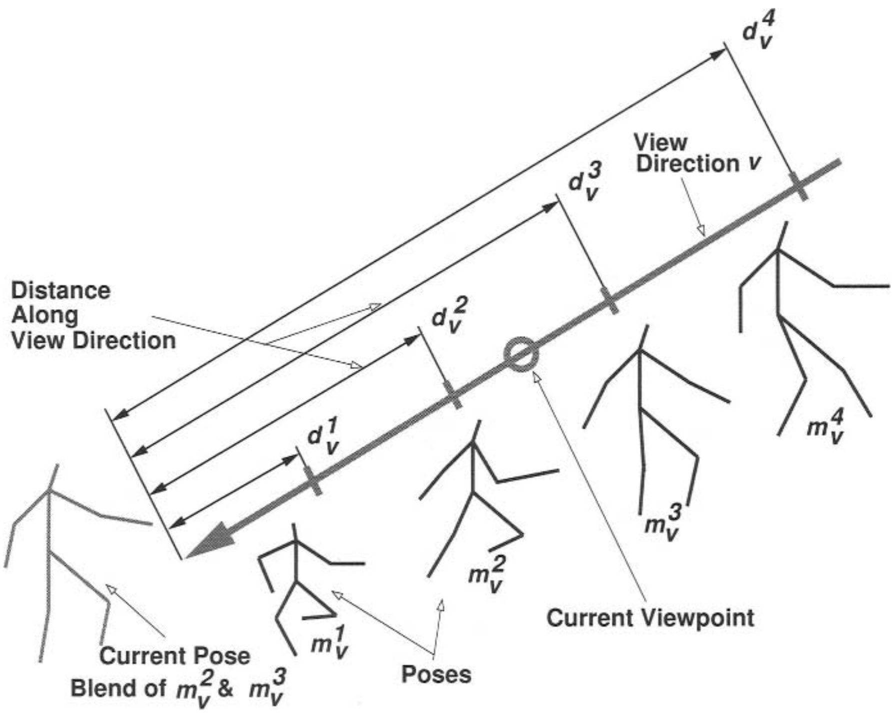


Fig. 2.14. Change of character pose with change of distance of the current viewpoint along a view direction.

Given a set of key viewpoints, v, and the associated tuple lists, (d_v^l, m_v^l), we want to generate an animation for a camera path, $P(\underline{v}, \underline{d}, \underline{t})$. The added parameter $\underline{d}$ is the distance of the viewpoint along the unit view direction $\underline{v}$. The vector $q_{\underline{v}} = \underline{d}\ \underline{v}$ gives the position of the current viewpoint (see Fig. 2.15). We determine the r-closest key viewpoints to $\underline{v}$ on the envelope as before. Now for every key viewpoint, $\bar{v}$, in the r-closest set of $\underline{v}$, we project the vector $q_{\underline{v}}$ on $\bar{v}$ and find the length of the projected vector. The projected length $\underline{d}\ \underline{v} \cdot \bar{v}$ is the distance $\underline{d}$ projected along $\bar{v}$. Find $d_{\bar{v}}^{l1}$ and $d_{\bar{v}}^{l2}$ from the tuple list of $\bar{v}$ such that $d_{\bar{v}}^{l1} \leq \underline{d}\ \underline{v} \cdot \bar{v} < d_{\bar{v}}^{l2}$. It is always possible to find a $\beta_{\bar{v}}$ such that

$$\underline{d}\ \underline{v} \cdot \bar{v} = (1 - \beta_{\bar{v}})d_{\bar{v}}^{l1} + \beta_{\bar{v}} d_{\bar{v}}^{l2} . \tag{2.2}$$

$\beta_{\bar{v}}$ locates a point, $q_{\bar{v}}$, along the corresponding $\bar{v}$ vector. The pose at each $q_{\bar{v}}$ is given by

$$m_{q_{\bar{v}}} = (1 - \beta_{\bar{v}})m_{\bar{v}}^{l1} + \beta_{\bar{v}} m_{\bar{v}}^{l2} , \tag{2.3}$$

where $m_{\bar{v}}^{l1}$ and $m_{\bar{v}}^{l2}$ are the poses associated with $d_{\bar{v}}^{l1}$ and $d_{\bar{v}}^{l2}$. Then the pose corresponding to the current viewpoint $q_{\underline{v}}$ is given as a weighted blend of the pose at each $q_{\bar{v}}$, as

$$m_{q_{\underline{v}}} = \sum_{\bar{v}} w_{q_{\bar{v}}} m_{q_{\bar{v}}} \, , \tag{2.4}$$

where $w_{q_{\bar{v}}}$ are the weights used for the blending. The process is shown schematically in Fig. 2.15. If the tuple list of $\bar{v}$ is a singleton, then it means that only one pose is available along that view direction at some distance. In such a case, $l = 1$ and the associated mesh, i.e., the same $m_{q_{\bar{v}}}$ is used for blending whenever $\bar{v}$ lies in a r-closest set.

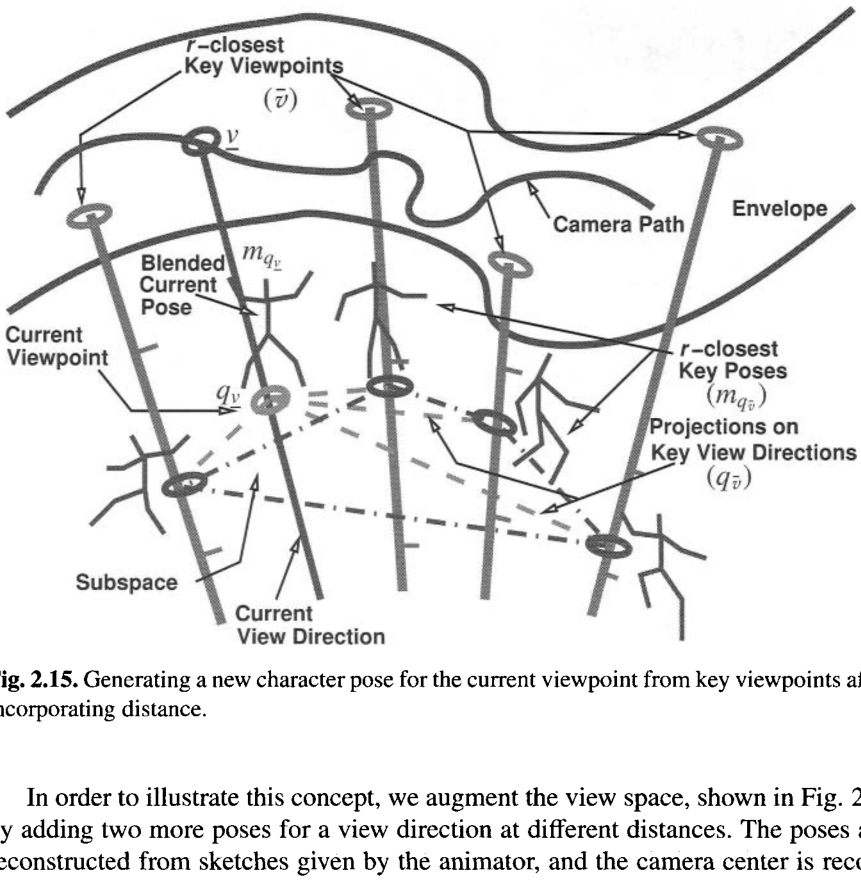


Fig. 2.15. Generating a new character pose for the current viewpoint from key viewpoints after incorporating distance.

In order to illustrate this concept, we augment the view space, shown in Fig. 2.9, by adding two more poses for a view direction at different distances. The poses are reconstructed from sketches given by the animator, and the camera center is recovered along with the distance of viewing (see Section 3.6.1). Two camera positions at different distances with their associated character poses are shown in Fig. 2.16. Now we trace another path for the rendering camera, specifying d_p for all points on the path, and the required animation is generated as explained above. The path traced is shown in Fig. 2.17. This also illustrates that there exist other paths that are capable of generating interesting animations. The framework can generate animation in real time as the animator traces out a path on the view space, thus making it possible for the animator to explore the view space very easily.

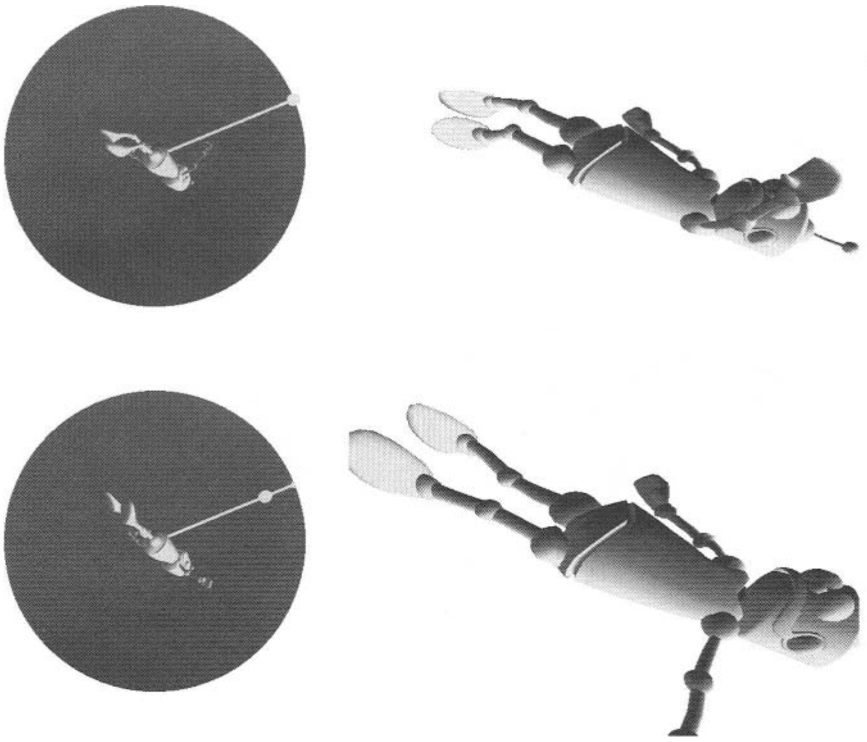


Fig. 2.16. On the left the two camera positions are shown — note that they only differ in the distance from the character and not the view direction. On the right the corresponding character pose is shown, as seen from their associated cameras.

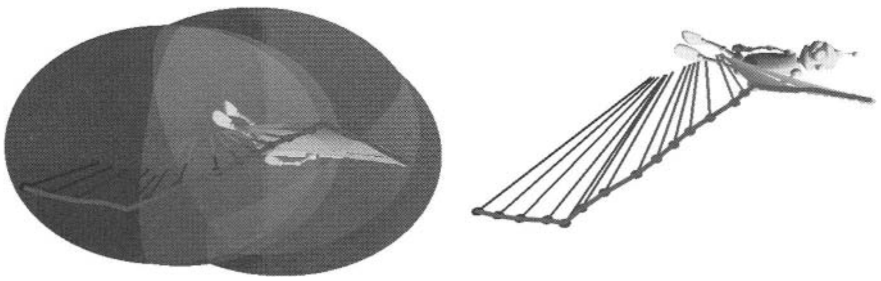


Fig. 2.17. The new path with distance variations along with the envelope of the view space.

Thus, in this framework we incorporate both the view direction and the distance of a viewpoint. It is fairly simple to incorporate changes in the character pose with changes in focal length of the camera in a manner similar to the one used for distance of the viewpoint. Hence, we capture all the different variations possible while defining a camera in the framework. The view direction, viewpoint, and focal length of the camera are the parameters that constitute a camera matrix. Note that the view space is an abstract representation and can be easily used with the view parameters encoded in the form of a camera matrix. In Chapters. 3 and 4 we present techniques used to extract the various view parameters from the camera matrix, which are recovered from the given inputs.

2.4 Other Extensions

We can easily extend the framework to handle other commonly occurring scenarios during animation. We briefly examine two of them here:

- In Section 2.2 we represent the view space as a collection of view directions independent of the sampling order. We can, however, retain the sampling order with the view directions and the character poses. This allows us to generate animations where the action requires the view directions or character key poses to be considered in a particular order.
  Consider an example where the camera is moved back and forth along a single view direction while the character is completing a movement sequence. If the ordering information is not used, the character poses are sorted along the view direction based on the distance of the camera from the character. This will lead to poses from the forward camera movement being interleaved with poses from the backward movement. When the animation is generated by moving the camera along this view direction, the character pose will alternate between poses taken from the front and the back movement sequences. This can be easily avoided by using the ordering information.
- We have only considered moving-camera animations until now. It is, however, possible to represent an animation with a stationary camera in the framework. The character poses are ordered by key frames and associated with a constant view direction. The animation is generated using normal keyframing. It can be seamlessly blended with a moving-camera animation by simply maintaining the desired continuity among the cameras in successive frames. Hence, the framework can easily fit into a conventional animation work flow. We explain this in more detail in Section 3.6.3.

2.5 Chapter Summary

In this chapter, we first start by examined the prior work done toward developing different representations of character poses and rendering cameras. Animations are created using abstract representations for a space of character poses, like simplicial complexes and spatial keyframing. Different methods for representing the rendering camera have also been reported in the literature. These include multiprojection rendering, nonlinear projections, and cognitive controls for automatic cinematography. We provide a single representation that embodies the camera and character pose association and allows us to generate animations from it.

Next, we have discussed the related work, which are based on the idea of view dependence of the character pose. We start with the work on view-dependent geometry. We see that the technique introduces the idea of a geometry that changes with the viewpoint. Some work has also been done on generating 2D illustrations containing observer-dependent deformations. We present a comprehensive, semiautomatic authoring solution for view-dependent animations that is easy to use and efficient.

We have presented a theoretical framework that captures the rich diversity offered by view-dependent animations into a compact representation. Key viewpoints and associated key poses of the character provided by the animator form a view space. Any path traced on the envelope of this view space generates a view-dependent animation.

We also show how to incorporate distance of the viewpoint into the view space. Hence, we can generate animations where the pose of the character changes in response to changes in the distance of the viewpoint from the character. We can easily incorporate changes in the focal length of the camera, as a parameter to perform view-dependent animation, into the view space.

Using the envelope to characterize a view space allows us to have a better understanding of the concept of staging actions while generating a view-dependent animation. In subsequent chapters we present the techniques used to implement this framework and to create view-dependent animations from multiple types of input like sketches and videos.

3

View-Dependent Animation from Sketches

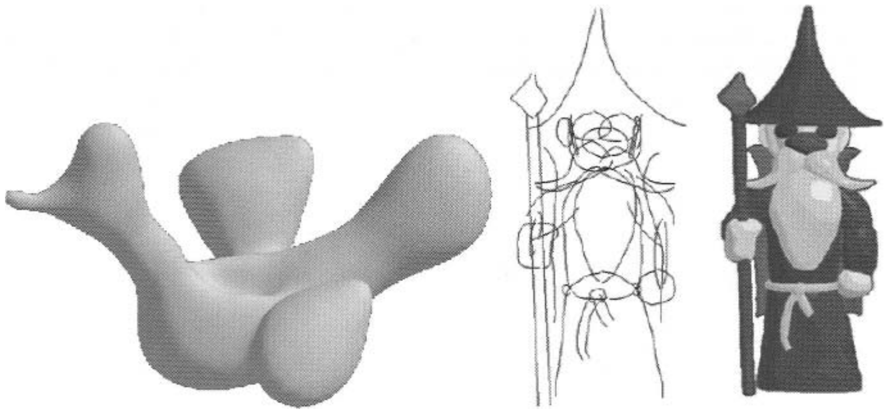
In Chapter 2, we introduced a framework for representing view-dependent animations. The view space (see Section 2.2) captures all the information necessary to generate a view-dependent animation. This view space, however, has to be physically realized from the inputs available to the animator. In this chapter we present the use of sketches to create such a view space and generate a view-dependent animation [25]. Before we explain this technique, we discuss the prior work that exists in the area of creating animation from sketches.

3.1 Prior Work

There have been numerous attempts toward designing systems that try to retain the expressivity and ease of creation of a 2D sketch while allowing generation of 3D animations and character models from it. Sketches have been used for creating, posing, and animating 3D models of characters. We first look at the various attempts made toward creating character models from a sketch.

3.1.1 Creating character models from sketches

The SKETCH [143] system combines mouse gestures and simple geometric recognition to create and modify 3D models. It uses a gesture grammar to create simple extrusion like primitives in orthogonal view. The Teddy [66] system presents techniques for modeling from sketches, i.e., given a drawing, the system tries to recreate a geometric description of the scene. The system allows creating a surface by inflating regions defined by closed strokes. Strokes are inflated so that portions of the mesh are elevated based on their distance from the stroke's chordal axis. Teddy also allows users to create extrusions, pockets, and cuts to flexibly edit the models. The limitation of SKETCH and Teddy, however, is that the inferred geometry is often incorrect, and these errors become more and more apparent with changes in the viewpoint. In a later work, Igarashi and Hughes [65] present in SmoothTeddy, a technique to refine the irregular polygonal meshes resulting from the original Teddy algorithms. A beautification process, based on the Skin algorithm [98], generates near-equilateral triangles with a near-uniform distribution of vertices on the surface to hide irregularities in the original polygonal model. It is then refined to generate a dense polygonal mesh that smoothly interpolates the beautified mesh [see Fig. 3.1(a)]. Cherlin et al. [26] also present a sketch-based system for the interactive modeling of a variety of free-form 3D objects using just a few strokes. It draws on conventional drawing methods and work flow to derive interaction idioms that should be familiar to illustrators. They develop algorithms for parametric surfaces using rotational and cross-sectional blending. The system allows the modeling of small, simple parts of a character. The user then directs the assembly of these parts using standard techniques like translation and rotation by clicking and dragging with the mouse. An example of a character created by such an assembly is shown in Fig. 3.1(b).


(a) A bird designed using SmoothTeddy (image courtesy Igarashi and Hughes [65])

(b) A wizard designed using system developed by Cherlin et al. (image courtesy Cherlin et al. [26])

Fig. 3.1. Examples of character models made from sketches.

Cohen et al. [28] present Harold, a system that allows an artist to draw on the image plane and thereby express a stylized 3D world. They make simplifying assumptions about the underlying geometry. A billboard is used as the primitive geometric structure to model the scene. A billboard is typically a plane with an image texture mapped onto it. This plane rotates about some point or axis to face the viewer as much as possible. When the user draws a stroke over a billboard, their system simply projects the stroke onto the billboard and stores it; then, in order to display the billboard, they rerender each stroke, rotated appropriately. Objects in Harold maintain the distinct stylistic appearance and subtleties imparted by the user, and its worlds, thus maintain their intended style and character as the viewpoint changes.

Tolba et al. [127] present a drawing system for composing and rendering perspective scenes. They use projective 2D points to compose various renderings of a scene and support perspective drawing guides, 3D-like viewing and object manipulation, scene illumination and shading, and automatic shadow construction. The 2D representation, however, has limited use for 3D animation. Sýkora et al. [122] present an example-based framework for computer-assisted cartooning. They design new characters and poses by combining fragments of original artwork. The user can simply select an interesting part in the original image and then adjust it in a new composition using a few control scribbles. The method works on images as input, and thus, the cartoons are generated in 2D.

Another approach is to introduce 3D information into a 2D animation system by manual ordering of layers [96] or by underlying simplified 3D models like stick-figure skeletons [105]. Ženka and Slavík [130] present a system that when given a 2D sketch, creates a 3D polygon mesh describing the skeleton. It thus creates a hybrid sketch that can be rotated like a 3D object. The user can see the object from various angles without having to create a full 3D model or drawing the object again for each required view.

3.1.2 Posing character models from sketches

We want to pose the character's model in order to match it with the character's pose in the sketch. We do not want to create the model from the sketch. Early attempts at posing stick skeleton figures from sketches are made by Sabiston [113]. This system uses a simple reverse projection algorithm to reconstruct a skeleton pose from the sketch. The projection considered in this system is orthographic, and it resolves the depth ambiguity by user interaction. In another work, Hecker and Perlin [55], develop a sketch-based animation system using a touch-sensitive tablet. Their system, however, relies completely on the artist to resolve any ambiguity.

In a recent work, Davis et al. [36] provide a simple sketching interface for articulated figure animation. The user draws the skeleton on top of the 2D sketch. Then they reconstruct the various alternative poses of a 3D stick figure corresponding to the 2D pose, using techniques given by [87] and [123]. The user is allowed to pick the desired pose and perform a few corrections to it. This interface is supported by pose reconstruction and optimization methods specifically designed to work with imprecise hand-drawn figures. The system provides a simple, intuitive, and fast interface for creating rough animations that leverages the users existing ability to draw. The resulting keyframed sequence can be exported to commercial animation packages for interpolation and additional refinement. The skeleton posing technique used here is in spirit similar to the one used in our framework (see Section 3.5).

In another contemporary work, Li et al. [91] present a method for stylizing animations through drawings. They allow an animator to modify frames in the rendered animation by redrawing the key features such as silhouette curves. These changes are then integrated into the animation. To perform this integration, they divide the

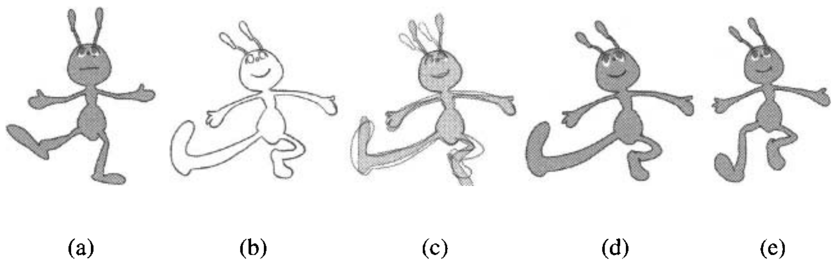


Fig. 3.2. (a) The motion captured movement is stiff and lacks personality, (b) example image was drawn by the animator to better express the character's personality, (c) motion editing matches the pose of the character is closer to the example image, (d) after layered warp, the mesh is warped to the example drawing's shape, (e) for a subsequent frame, the warping field is propagated (images courtesy Li et al. [91]).

changes into those that can be made by altering the skeletal animation and those that must be made by altering the character's mesh geometry. To propagate mesh changes to other frames, they use a modified image-warping technique that takes into account the character's structure. In this paper, the skeletal deformations are obtained by manually modifying a standard motion capture stream. It relies on the animator to find the frame in the original rendered animation that best matches the sketch and then creates a deformation field to warp the silhouette by matching the curves in 2D. The process is illustrated through an example in Fig. 3.2.

3.1.3 Animating character models from sketches

Some work has also been done on the creation of animation using sketch based interfaces. In an early work, Librande [93] presents a system which learns an example space from a set of vector drawings. It can then produce the in-between frames by constructing an interpolation function on the example space. The animation generated by this system is in 2D.

Thorne et al. [125] demonstrate motion doodles, a sketching interface for generating character motion. A continuous sequence of lines, arcs, and loops are parsed and mapped to a parameterized set of output motions that reflect the location and timing of the input character sketch (see Fig. 3.3). The system supports different types of motions in 2D and a subset of them in 3D. It is useful for fast creation and quick experimentation with various kinds of character motion. However, it cannot resolve ambiguities introduced by 3D mapping, and it is not suitable for animations that require unique or detailed motions.

Mao et al. [97] present an interface for sketching out rough 3D stick figure animation. The system allows the users to draw stick-figures with automatic figure proportion control. It utilizes figure perspective rendering, and it introduces the concept of thickness contrast as a sketch gesture combined with some other constraints

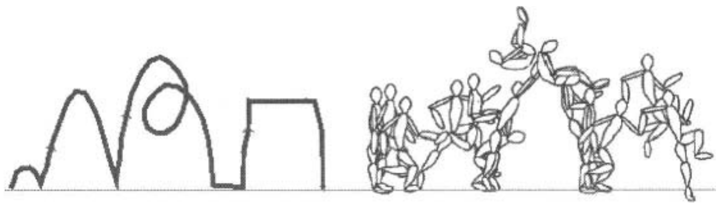


Fig. 3.3. A 2D motion sketch and the resulting animation (step, leap, front-flip, shuffle, hop) created using the motion doodles system (images courtesy Thorne et al. [125]).

or assumptions for pose recovery. The resulting pose can be further corrected based on physical constraints of the human body. Once a series of 3D stick figure poses is obtained, the user can sketch out motion paths The resulting 3D animation can be exported to VRML.

To summarize, we find that purely geometric approaches for creating 3D models from sketches like [66] and [143] suffer from a fundamental drawback — not all 2D drawings of a character can actually be generated from one 3D model. Dynamic view-dependent models are an ideal solution for this problem. In fact, the variations present in 2D drawings that cannot be captured by a conventional geometric 3D model (like a triangular mesh model) is one of the prime motivations behind using a view-dependent model [109].

We want to use sketches of a character to extract camera or view parameters from it and to pose the 3D model of the character, in order to create a view space of the recovered views and their associated character poses. Existing works discussed above often work only with scaled orthographic cameras and have no explicit notion of a camera recovery. We recover the best full projective camera and view direction that matches the sketch (see Section 3.4). We show that this flexibility in the recovery of the camera allows for dramatic effects, such as close-up shots, in the resulting animation. In our case, the camera recovery technique requires minimal user interaction to specify correspondences. This technique also allows the user to correct or refine the automatically reconstructed poses. We, further, match the deformation of the character's mesh to the sketch, which cannot be recovered by matching the skeletal pose only.

We have developed a pipeline to generate a view space from a set of sketches, which in turn allows us to generate a view-dependent animation. In the next section, we present an overview of this pipeline.

3.2 Overview of the Pipeline

Our technique for view-dependent animation generates a view space from a set of sketches using the pipeline we have developed (shown in Fig. 3.4). The animator provides a set of sketches as input. We assume that the 3D base mesh model of the character is also given. The pipeline processes one sketch at a time.

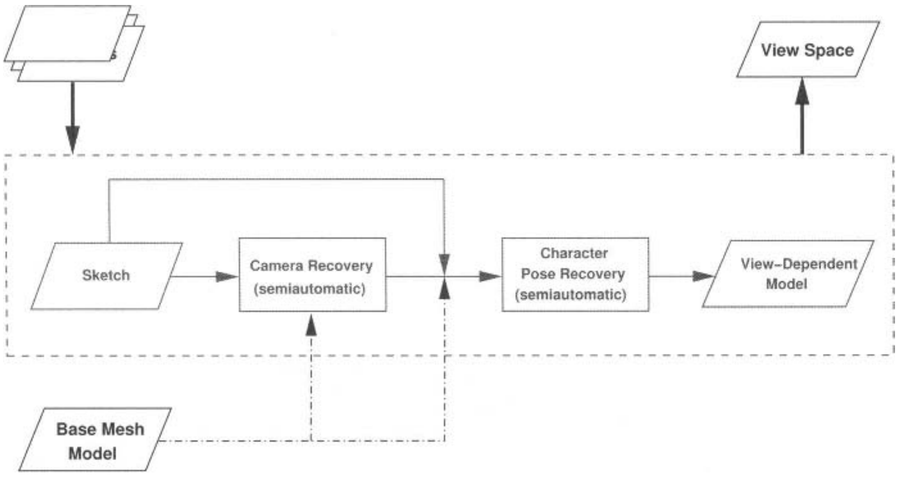

View Space
Sketch
Camera Recovery (semiautomatic)
Character Pose Recovery (semiautomatic)
View-Dependent Model
Base Mesh Model

Fig. 3.4. Schematic diagram depicting the pipeline to create a view space from a set of sketches.

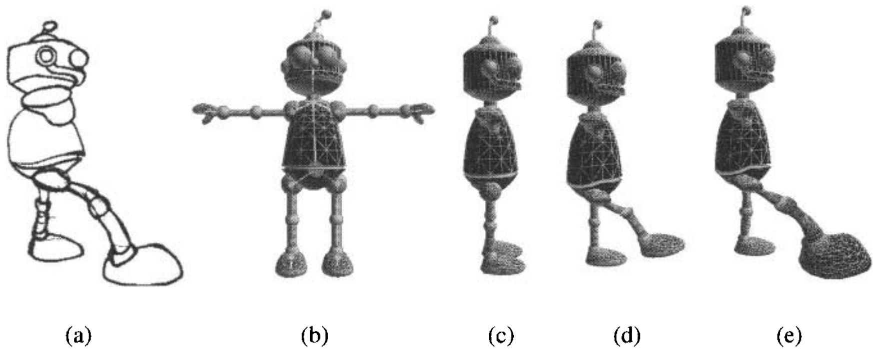


Fig. 3.5. Creating a view-dependent model from a sketch: (**a**) input sketch, (**b**) base mesh model, (**c**) recover a camera to orient the base mesh, (**d**) reconstruct the skeleton pose, and (**e**) deform the mesh to find the best possible match with the sketch.

Figure 3.5 shows the various stages of the pipeline using an example sketch and a base mesh model as input. We first align the viewpoint with the intended view direction in the sketch [see Fig. 3.5(c)] by recovering the camera using computer vision techniques. Then, we find the best match of the pose of the character[1], when seen from the recovered view direction, with the sketch by moving the skeleton embedded inside the mesh. The mesh model thus obtained is called a *posed mesh model* [see Fig. 3.5(d)]. The final stage deforms the mesh to match the silhouette of the model, as seen using the recovered camera, with the sketched character. The final mesh model is called a *deformed mesh model* [see Fig. 3.5(e)]. The final character pose, output at the last stage of the pipeline, together with the recovered camera is called the *view-dependent model.* Note that the camera recovery and the character pose recovery are semiautomatic processes and require some interactive inputs. This process is repeated for every sketch, and all the resulting view-dependent models together form the view space.

We now describe this process in detail in the following sections. To bootstrap the process, we need an additional set of inputs, which are created as described in the next section.

3.3 Inputs

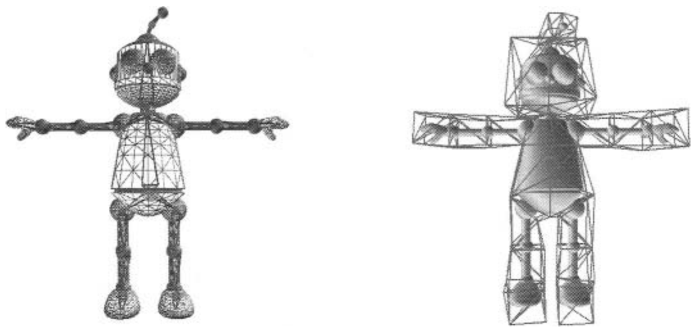
The primary inputs to the pipeline are a sketched pose [see Fig. 3.5(a)] provided by the animator and a base mesh model [see Fig. 3.5(b)]. The base mesh model is a 3D mesh model of the character to be animated, made using any modeling software. We embed a skeleton into the mesh and enclose the mesh in a lattice to facilitate the automated character posing process (explained in Section 3.5). We describe the methods for creating these.

3.3.1 Interactive skeleton and lattice construction

We have implemented a simple interactive technique for skeleton construction inspired by Capell et al. [24]. It allows a skeleton, embedded in the mesh, to be constructed interactively in just a few minutes. The user creates a joint by clicking on the object with the mouse. If the ray through the clicked point (from the camera projection center) intersects the object at least twice, a joint is placed midway between the first two intersections. This positioning scheme produces joints that are centrally located inside the object. Two joints are selected to define a bone. When the whole skeleton has been created, the user selects a joint as the root, and a transformation hierarchy is created automatically. We give an example of a skeleton embedded in the mesh using this technique in Fig. 3.6(a).

The skeleton need not correspond to an anatomically valid skeleton (in fact the object may not even have a skeleton, for example, an inanimate object like a guitar or a lamppost). The skeleton is a control mechanism provided to the animator to help define the pose of the character with ease. The user has complete flexibility and control over the way the skeleton is defined.

[1] See Section 2.2 for a definition of *pose* as used in this book.



(a) Skeleton embedded inside the mesh model (b) Lattice enclosing the mesh model

Fig. 3.6. Example input for the pipeline.

The mesh is also enclosed in a lattice. The lattice is made up of tetrahedral cells and encloses the base mesh model. The lattice is defined in a manner such that each lattice cell is associated with one skeleton bone. Thus the lattice construction is based on the way the underlying skeleton has been defined. We give an example of such a lattice in Fig. 3.6(b). Further details about the lattice and skeleton construction can be found in [70].

We now present the various steps of the pipeline in detail and explain how we create view-dependent animations from sketches.

3.4 Recovering the Camera

The first step in the pipeline (see Fig. 3.4) is camera recovery. In order to pose the character as drawn in the sketch, we need to first recover the intended view direction from the sketch. We describe the process for a single sketch in which we find a camera such that the projection of the character's model using the recovered camera is aligned to the sketched pose [see Fig. 3.5(c)]. This process is repeated for every sketch the animator provides.

The animator usually provides the sketches on paper. Then they are scanned into the computer. The pipeline works on these scanned sketches. The user clicks correspondences between the sketched pose and the skeleton joints. The skeleton joints are marked on the sketch. We can handle two broad categories of sketches. For sketches of the character, correspondences are very easy to specify as the user can locate the joints on the sketch easily and accurately [see Fig. 3.7(a)]. The second category of sketches that we can handle are rough *mannequin* sketches [see Fig. 3.7(b)]. Such sketches are easier and faster to draw and are often used for rough pose planning during early stages of the animation (as is also shown in Fig. 1.1). If the bone proportions of the mannequin are roughly the same as that of the character, then locating the skeleton joints on such a sketch is quite intuitive for the animator and correspondences can be approximately specified very quickly. The camera recovery engine is robust enough to determine a feasible camera for the approximately placed joints and works equally well with both these categories of sketches. We demonstrate the use of mannequin sketches in the *Olaf Reloaded* animation, where the whole animation has been generated from such sketches.

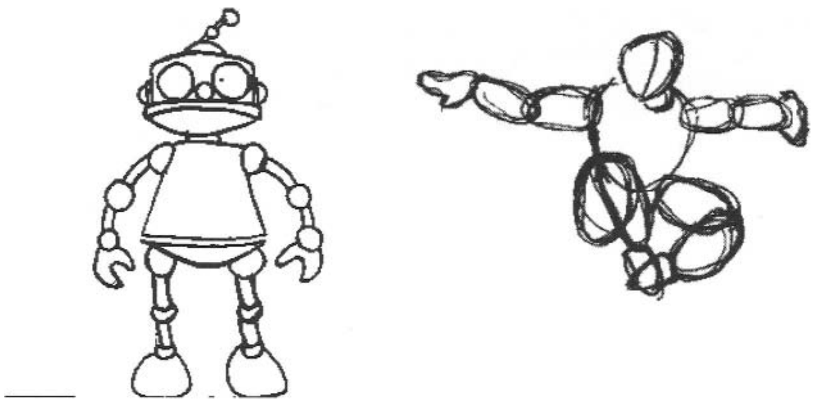
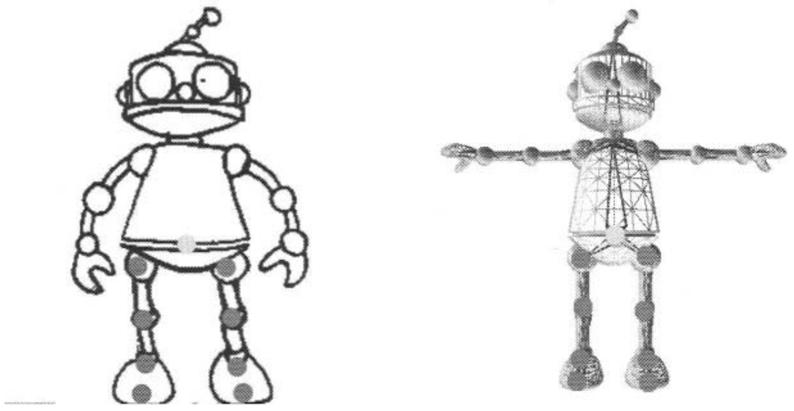

(a) A sketch of a character (b) A sketch of a mannequin

Fig. 3.7. Possible input sketch types.

The joints used during camera recovery have to be rigid relative to a change in pose i.e., the joints must not have moved from the base mesh model to the posed mesh model. The user needs to click the position of these rigid joints (see Fig. 3.8). The first point marked on the image must correspond to the root of the skeleton. This repositions the image coordinate system origin accordingly (as the root forms the origin of the skeleton's coordinate system). The minimum number of joints whose positions must be clicked on the sketch can vary from 3 to 6 depending on the type of the camera to be recovered. An orthographic camera has only five degrees of freedom and hence requires only three point correspondences (see Section A.3). A full projective camera has eleven degrees of freedom and hence, requires six point correspondences if it is to be determined completely. Subsequently the two point lists (2D sketch points and 3D skeleton joints) are normalized. The full projective camera is computed using the Normalized Direct Linear Transformation Algorithm and the affine camera is computed using the Gold Standard Algorithm (for further details see Appendix A and [52]). These are numerically robust techniques and work well with



(a) A sketch with rigid joints marked (b) The corresponding joints marked on the skeleton

Fig. 3.8. Marking of joint correspondences.

hand-clicked correspondences. The weak perspective and orthographic cameras are recovered using techniques similar to those used for affine cameras. In most situations, we have worked with the full projective and affine cameras. The full projective camera is better suited for cases where close-ups of the characters are required (i.e., the distance of the camera from the character is comparable to the width of the character along the view direction). The affine and other cameras are better for cases where the camera is further away from the character (i.e., the distance of the camera from the character is considerably more than the width of the character along the view direction) and hence the perspective foreshortening effect is not pronounced. The full projective camera is of the form

$$\mathbf{P} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} . \tag{3.1}$$

The camera thus estimated projects the clicked joints on the corresponding points on the sketch. If the camera recovered is **P** and the camera center is **C** then we must have

$$\mathbf{P} \cdot \mathbf{C} = 0 . \tag{3.2}$$

So the camera center is recovered as the right null space of the camera matrix. For the full projective camera we get a finite camera center, but in case of the affine, weak perspective, and orthographic cameras the camera center is a point at infinity. In such cases, however, we are only interested in the camera view direction, i.e., the unit vector in the direction of the line joining the camera center with the look-at point, and this is recoverable in all cases. The anatomy of the projective camera is given in Section A.2.

Some care has to be taken while choosing the rigid joints. All the joints should not lie nearly on the same plane. If this is the case, then the solution returned by the algorithm is unique modulo a flip about the image plane. The principal axis vector is a vector along the principal axis of the camera, directed toward the front of the camera. This can be used to detect the condition when the camera recovered is actually behind the image plane. Such cases are easily corrected either by clicking a few out-of-plane joint correspondences and recomputing or by simply flipping the previously obtained solution. If the requisite number of rigid joints cannot be identified, vertices of the mesh, satisfying the rigidity requirement, can be used for point correspondences.

When we look toward the 3D character using the recovered camera, it appears aligned with the sketched pose. We refer to this recovered viewpoint as a *key viewpoint*. Next we deform the mesh model and change its pose to match the sketch.

3.5 Posing the Character

In this section we describe the process of posing a character from a single sketch and obtaining a view-dependent model. We use a two-layered deformation engine for this purpose. In the first layer we match the skeletal pose of the character with the sketch, to obtain the posed mesh model, using a view-dependent posing algorithm (see Sections 3.5.1 and 3.5.2). In the second layer we match the silhouette of the mesh model of the character (in the recovered view) with the sketch, to obtain the deformed mesh model, using a view-dependent mesh deformation algorithm (see Sections 3.5.3 and 3.5.4). We look at both the layers one by one.

3.5.1 Skeleton-based posing

We first match the overall or gross level pose of the sketched character. This can be done by changing the articulation of the skeleton embedded in the mesh.

Inverse kinematics

In order to pose the skeleton we make use of inverse kinematics (IK). Inverse kinematics is most commonly used to interactively pose articulated characters. We want to automate the posing process as much as we can, but we also want to let the animator have interactive control. This way the animator can manually tweak the automatically recovered poses if so desired. Other variants of IK, like style-based IK [49] and mesh-based IK [121] may be used to enhance performance of the posing algorithm we present. This, however, would require no change in the pipeline or the posing algorithm we present later (see Section 3.5.2).

A skeleton is modeled as a collection of rigid objects (also called *links* or *bones*) connected by joints. The skeleton forms a rooted tree, with a special joint marked as the root. We define a *kinematic chain* as any sequence of joints in this skeleton such that the sequence does not span across branches in the tree. We define the starting joint of this kinematic chain as the *base* joint for the chain and the far-end (distal) joint as the *end-effector*. Note that the end-effector can even be a joint that is not a leaf, as the chain can start and end anywhere as long as it does not span a branch. The base is fixed and cannot move, while the end-effector is free to move.

Given a vector $\mathbf{q}$ of known joint variables, the *forward kinematic* problem of computing the position and orientation vector $\mathbf{x}$ of the end-effector is simple to solve, and has the form

$$\mathbf{x} = f(\mathbf{q}) . \tag{3.3}$$

But if the goal is to place the end-effector at a specified position and orientation $\mathbf{x}$, then determining the appropriate joint variable vector $\mathbf{q}$ to achieve the goal requires a solution to the inverse of Equation (3.3),

$$\mathbf{q} = f^{-1}(\mathbf{x}) . \tag{3.4}$$

Solving this *inverse kinematic* problem is not simple. The function f is nonlinear, and the inverse mapping of Equation (3.4) is not unique — there may be many $\mathbf{q}$'s for a given $\mathbf{x}$. A natural approach is to linearize the problem about the current chain configuration. Then the relationship between the joint velocities and the velocity of the end-effector is

$$\dot{\mathbf{x}} = J(\mathbf{q})\dot{\mathbf{q}} . \tag{3.5}$$

If $J = \partial f / \partial q$ is the Jacobian matrix, then the inverse relationship becomes

$$\dot{\mathbf{q}} = J^{-1}(\mathbf{q})\dot{\mathbf{x}} . \tag{3.6}$$

Implementation

We use an exponential map parametrization for joint rotations as given by [48]. Every nonzero vector in $\mathbb{R}^3$ has a direction and magnitude. We can associate a rotation with each vector by specifying the direction as an axis of rotation and the magnitude as the amount of rotation. If we augment this relationship by associating the zero vector with the identity rotation, the relationship is continuous and is known as the exponential map (see Appendix B for more details on the exponential map). The exponential map is used as it does not require repeated normalization (like quaternions) to stay in a meaningful subspace, and it results in smaller dimension state vectors, which leads to faster performance. It is simple to compute the Jacobian at a node in a transformation hierarchy with respect to all the end-effectors below it in the hierarchy. This is used in the inner loop of the inverse kinematics solver and hence needs to be fairly fast. The exponential map turns out to be a very good choice for this purpose, as it allows the inverse kinematics posing mechanism to respond at interactive rates.

Once the Jacobian is obtained, we can solve for the change in the joint state vector. Here we find a least-squares solution using the pseudoinverse of the Jacobian.

We compute the pseudoinverse of the Jacobian using the Singular Value Decomposition (see [47]). However, near a singularity, the problem becomes ill-conditioned, and the norm of resulting least-squares solution may tend to infinity. So it needs to be regularized, which we do using *damped least-squares* [126]. Once the pose of a chain has been computed, the state vector for the chain is updated appropriately and the change in joint transformations is propagated throughout the skeleton to pose the skeleton. A detailed description of the techniques mentioned above may be found in [13] and [136].

Inverse kinematics finds the best possible kinematic chain configuration to reach the specified goal by iteratively searching in a space of possible solutions. Chain configurations where the motion at any of the joints exceeds permissible limits are not valid solutions and are discarded to prune the solution search space. This is done using joint limits. Usually, joint limits are specified as constraints in IK. We have implemented spherical joint limits using *joint reach cones*. It is a natural representation of the range of allowed motions for an articulated body segment, borrowed from biomechanics. We have used the techniques of Wilhelms and Van Gelder [137] for specifying and enforcing joint limits in IK using reach cones. Reach cones can be specified interactively, and can be turned off if the animator so desires. Sometimes animated character's have range of motions that are exaggerated when compared with a normal human. Since reach cones give a visual representation of the joint limits being enforced, they are more easily specified than other joint constraint techniques, even in unconventionally moving characters. Reach cones can be efficiently implemented and have no discernible effect on interactive speeds of the IK posing system. Reach cones are explained in greater detail in Appendix C.

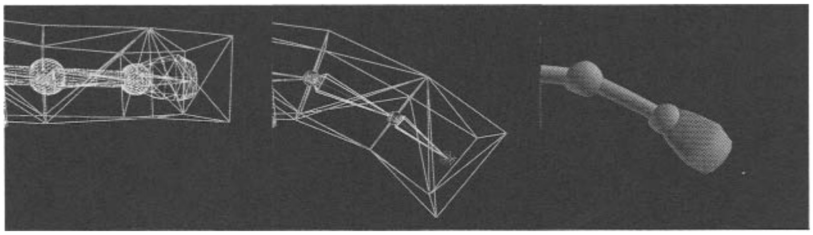


Fig. 3.9. Blended skinning: (**a**) a part of the mesh with the embedded skeleton and the lattice, (**b**) deformation in the lattice as the skeleton articulation changes, (**c**) resulting deformation of the mesh due to (**b**).

Inverse kinematics poses the skeleton. For posing the mesh model, the mesh is made to move with the skeleton using in essence a *blended skinning* [83, 84] approach. We use the lattice that encloses the mesh to do this. During construction, every cell of the lattice is associated to some skeleton bone. The cell vertices may be shared among two or more cells. So the cell vertices have weights assigned to them that denote the degree of influence of the corresponding underlying skeleton bones on those vertices. Every lattice cell vertex also has its own coordinates recomputed in the local coordinate system of their associated bones. When the underlying bones move, the new position of the lattice cell vertex is computed using *bones blending* [77]. Every lattice cell is also associated with all the mesh vertices contained inside the tetrahedral lattice cell. Every mesh vertex is associated to at most one lattice cell. To move the mesh, the new positions of the mesh points contained in a lattice cell are calculated using their barycentric coordinates. This results in a smooth deformation of the mesh as the skeleton is moved (see Fig. 3.9).

Using only IK, however, requires extensive user interaction to pose the mesh. We propose a novel *view-dependent posing algorithm* to pose the 3D mesh model of the character.

3.5.2 View-dependent posing algorithm

Interactive posing using IK though possible, requires extensive user intervention. The number of degrees of freedom that the user has to manipulate can seem overwhelming. Thus we propose an algorithm that restricts IK to find solutions that are consistent with the recovered camera.

Before the algorithm can start, the user needs to specify the joint correspondences on the sketch for all the joints that need to be moved during the posing and that were not marked during the camera recovery phase (see Section 3.4). Now we define the goal as the desired 3D position of the end-effector of the kinematic chain. The algorithm now proceeds as given in Algorithm 3.1.

```
Require: The camera must be estimated before this algorithm can be run.
Require: Correspondences for all the joints to be deformed must be marked.
1 begin
2     repeat
3         Select a simple kinematic chain.
4         Back project 2D end-effector position as the 3D goal using the
          recovered camera.
5         Run IK to make the chain reach the goal.
6     until the desired pose has been achieved
7 end
```

Algorithm 3.1: View-dependent posing algorithm.

The user marks out a kinematic chain that needs to be deformed. For this purpose, the user only has to click on the joint names in a hierarchical graphical user interface (GUI) to identify the joints in the chain on both the sketch and the skeleton. The position of the end-effector is back projected from the sketch into 3D space using the pseudoinverse of the recovered camera (see Fig. 3.10). This is done in the following manner.

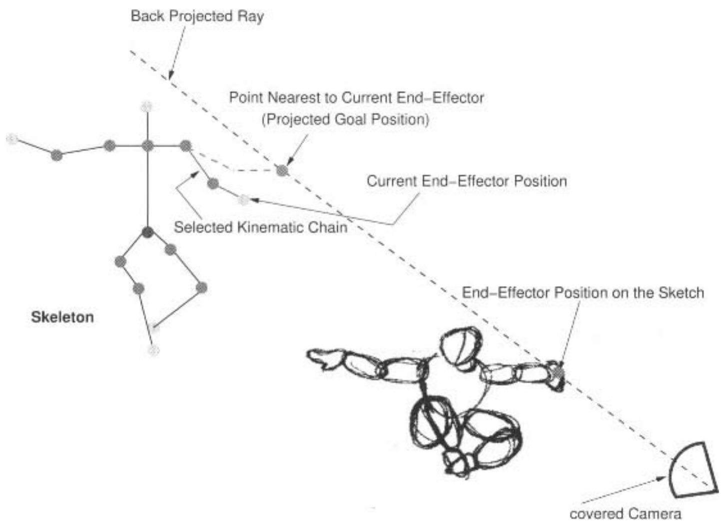

Back Projected Ray
Point Nearest to Current End-Effector
(Projected Goal Position)
Current End-Effector Position
Selected Kinematic Chain
End-Effector Position on the Sketch
Skeleton
covered Camera

Fig. 3.10. Back projecting the goal using the recovered camera.

Let the end-effector position on the sketch be $\mathbf{e} \equiv (x_e, y_e)$ and the recovered camera be $\mathbf{P}$ [$\mathbf{P}$ is of the form given in Equation (3.1)] with the camera center as $\mathbf{C}$, then projecting $\mathbf{e}$ back using $\mathbf{P}^+$ (the pseudoinverse of $\mathbf{P}$) gives us a 3D ray $\mathbf{R}_e$,

$$\mathbf{R}_e = \mathbf{C} + \lambda\,(\mathbf{P}^+ \cdot \hat{\mathbf{e}})\,, \tag{3.7}$$

where $\hat{\mathbf{e}}$ is $\mathbf{e}^\mathsf{T}$ expressed in homogeneous coordinates, i.e., $\hat{\mathbf{e}} \equiv (x_e, y_e, 1)^\mathsf{T}$.

Now we find the point (X, Y, Z) on this ray that is closest to the current position of the end-effector joint in 3D. If the current end-effector position is given by $\mathbf{E} \equiv (X_e, Y_e, Z_e)$, then the new position of the end-effector $\mathbf{E}_{new}$ is given by the solution of the following minimization:

$$\min_{(X,Y,Z)} = (X - X_e)^2 + (Y - Y_e)^2 + (Z - Z_e)^2$$

subject to

$$\begin{aligned} (p_{11} - p_{31}x_e)\,X + (p_{12} - p_{32}x_e)\,Y + (p_{13} - p_{33}x_e)\,Z + p_{14} &= p_{34}\,x_e\,, \\ (p_{21} - p_{31}y_e)\,X + (p_{22} - p_{32}y_e)\,Y + (p_{23} - p_{33}y_e)\,Z + p_{24} &= p_{34}\,y_e\,. \end{aligned} \tag{3.8}$$

$\mathbf{E}_{new}$ is the goal position in 3D. The constraints arise from the fact that $\mathbf{P}$ must project the point (X, Y, Z) to $\mathbf{e} \equiv (x_e, y_e)$ on the sketch. So the constraints embody the camera projection equation. Once we have projected the goal, the IK layer takes over and deforms the chain in 3D to make it reach the goal (see Fig. 3.10). Since IK is tied to the camera recovered and is guided by it to find a pose that satisfies the view-dependent constraint, the projection of the posed mesh model matches (in terms of the skeletal pose) the sketch.

The algorithm is numerically stable and works at interactive rates. Solving IK involves inverting the Jacobian using the Singular Value Decomposition and regularizing the solution near singularities using damped least squares (as explained in Section 3.5.1). The minimization to enforce the projection constraints can be efficiently solved as a constrained optimization by using Lagrange multipliers. The performance of the algorithm is dependent on the accuracy of the camera recovery phase. The algorithm works better with the joint reach cones turned on, as then the solution is better constrained.

The overall configuration of the chain achieved by the algorithm may not be the one desired by the animator. Often the easiest way around this is to consider short chains (i.e., chains with up to 3 segments). In all the experiments we performed, the algorithm always found the desired chain configuration for short chains. Short chains may result in more number of chains being selected, even then the number of joints to be clicked are about 2 to 10. This is still easier than posing directly in three dimensions. The algorithm works best if the small chains starting from a fixed root are successively posed as we move out toward the end-effector. This strategy of chain selection can be programmed, making the algorithm completely automatic.

The algorithm always selects the point on the ray that is closest to the current end-effector position as the new goal. The reason for assuming this as the new goal position is that the algorithm chooses a point that causes least movement or change from the existing end-effector position. If there is a cost or energy associated with the distance moved by the kinematic chain, then the algorithm makes a choice that minimizes this energy. The pose reconstruction is unambiguous as it always finds the best possible pose, which minimizes the reprojection error from the recovered camera. If, however, the above process does not pose the chain to the animator's satisfaction then the animator can still correct the pose of the chain by tweaking the bone positions interactively using IK.

This process is repeated for every chain till the desired pose of the complete skeleton is achieved. In the current implementation the posing process poses a single chain at a time. This can be easily extended to simultaneous solution of multiple chains reaching for multiple goals using techniques given in [12]. The lattice deforms the mesh whenever IK repositions a chain. Thus at the end of the posing phase we have the *posed mesh model* [see Fig. 3.5(d)].

Further, we can pose elements of the character that may not be there in the sketch using the interactive posing facility available in the IK layer. For example, in one of our results the character has a tail, but the input sketches are mannequin sketches and do not have a tail in them; so the tail is interactively posed by the user.

The posed mesh models become the *key deformations* associated with the recovered cameras that are the *key viewpoints*, and these constitute the view-dependent models. It may be argued that these are just posed and not deformed, but the geometry of the mesh model has changed from the base mesh model (due to the lattice layer moving the mesh along with the skeleton), and animating using these models produces a view-dependent animation where the geometry responds to changes in view. We illustrate this with an animation sequence in the results. If, however, the mesh is further deformed using a view-dependent mesh deformation algorithm in order to make it match better with the sketched character, we get an improved view-dependent model.

3.5.3 Mesh deformations

Animators frequently tend to exaggerate or stylize deformations of animated characters (e.g., a character's head develops a large bump when he gets hit by a hammer). Such deformations are not the result of articulation and hence cannot be reproduced by just using IK. In order to model such deformations, the mesh is deformed using the technique of direct free-form deformation (DFFD) [64, 80]. We choose DFFD to perform the deformation because it is simple and efficient. Since it works with any general control lattice, and we have already defined a lattice enclosing the mesh, it is the most convenient choice. The lattice is made of tetrahedral cells. Since the tetrahedron is a simplex, it allows a linear barycentric basis. Using this linear parameterization for defining local coordinates inside the tetrahedron makes the DFFD computations extremely fast and efficient.

The free-form deformation (FFD) method deforms an object by first assigning to each of its points within the deformation lattice a set of local coordinates. Once the control points are moved, the new location of an object point is determined by a weighted sum of the control points. The weights are functions of the coordinates originally assigned to the point. Hence, a positional change of the control points changes the location of the points. Direct free-form deformation involves moving a set of selected points of the object to some target locations by determining the change in control point positions that will effect this change. Let $\mathbf{q}$ be the vector of points to be moved, and $\mathbf{S}$ be a vector of all the control points, then

$$\mathbf{q} = \mathbf{BS}\,, \tag{3.9}$$

where $\mathbf{B}$ is a matrix of all the blending functions. If $\mathbf{q}_{new}$ is the vector with the new positions of the points, then $\mathbf{q}_{new} = \mathbf{B}(\mathbf{S} + \Delta\mathbf{S})$ or

$$\Delta\mathbf{q} = \mathbf{B}\Delta\mathbf{S}\,, \tag{3.10}$$

where $\Delta\mathbf{S}$ is the change in the position of the control points and $\Delta\mathbf{q}$ is the change in position of the object point. We are given $\Delta\mathbf{q}$ (as the difference between $\mathbf{q}_{new}$ and $\mathbf{q}$), and we want to find $\Delta\mathbf{S}$ that satisfies Equation (3.10). This can be solved by computing the pseudoinverse of $\mathbf{B}$.

We use the lattice enclosing the character mesh (see Section 3.3) as a control lattice for DFFD. Our implementation allows the user to deform the mesh interactively by direct manipulation with mouse clicks. The user can move groups of points as well as individual mesh points. For deformation of the mesh consistent with the view, we have a view-dependent mesh deformation algorithm that couples the process of deforming the mesh with the recovered camera. We describe this algorithm below.

3.5.4 View-dependent mesh deformation algorithm

Manually moving mesh vertices to match them with the sketch is a very cumbersome process. Since such a matching is guided by only visual inspection, it is error prone and inaccurate. Hence, we propose an algorithm to restrict the solution space of DFFD by the projection constraints imposed by the recovered camera. This allows us to closely match the contours of the 3D mesh with the curves of the sketched character when the mesh is viewed using the recovered camera.

For the algorithm, the user first needs to mark correspondences by clicking points on the sketch and the posed mesh model. These points are the inputs to the algorithm. We call these points the *input points*. These points usually belong to the silhouette curve of the sketch as we want to match the silhouette of the sketch with the mesh. The points in 3D where these input points are to be moved are called the *target points*. Algorithm 3.2 describes the process. The back projection and target point selection

```
   Require: The camera must be estimated before this algorithm can be run.
   Require: Correspondences for all the input points to be moved must be
            marked.
 1 begin
 2     repeat
 3         The 2D points on the sketch are projected back in 3D space as a ray,
           using the recovered camera.
 4         The points on these rays that are closest to their corresponding points
           in 3D are chosen as the target points.
 5         The set of input points and target points are passed on to the DFFD
           layer, which moves the input points to the target points.
 6     until the desired deformation is achieved
 7 end
```

Algorithm 3.2: View-dependent mesh deformation algorithm.

is done in the same manner as described in Section 3.5.2 for the skeleton-based posing. In this case, $\mathbf{e} \equiv (x_e, y_e)$ become the 2D points on the sketch. $\mathbf{P}$ is again the recovered camera with the camera center as $\mathbf{C}$. If the input 3D point is $\mathbf{E} \equiv (X_e, Y_e, Z_e)$, the target point (X, Y, Z) can be found by solving Equation (3.8) (see Fig. 3.11). The constraints ensure that $\mathbf{P}$ projects the computed target point (X, Y, Z) to $\mathbf{e}$ on the sketch. When the input mesh points move to the target points, the DFFD algorithm recalculates the position of the affected lattice cell vertices keeping the

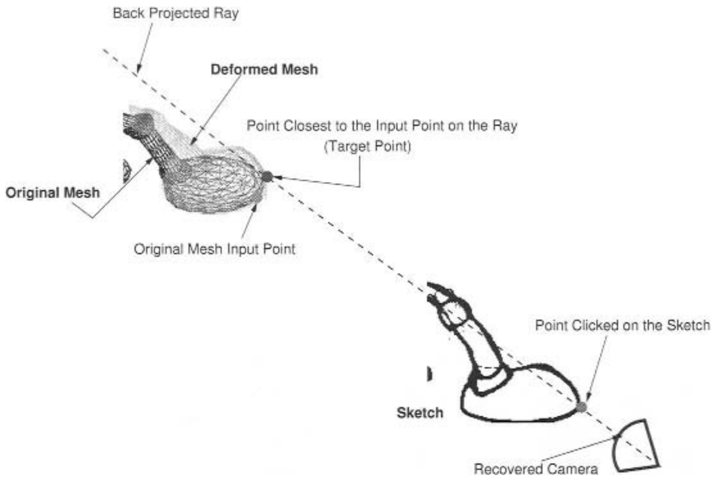

Back Projected Ray
Deformed Mesh
Point Closest to the Input Point on the Ray (Target Point)
Original Mesh
Original Mesh Input Point
Point Clicked on the Sketch
Sketch
Recovered Camera

Fig. 3.11. Back projecting the input mesh point using the recovered camera and deforming the mesh.

basis parameterization of the input mesh points constant. The rest of mesh vertices in the affected lattice cells suitably deform when the lattice cell vertices move.

The algorithm responds at interactive rates and is numerically stable. Direct free-form deformation computations involve the computation of the pseudoinverse of the matrix of blending functions. This is done using the Singular Value Decomposition. It can be proved that the pseudoinverse gives the least-squares solution to the DFFD problem (see Section 3.5.3). It is important that the input and target points are chosen consistently, i.e., the points in the neighbourhood of a point should move almost similarly to ensure smooth variation of the surface of the mesh. It is also important that points that should not be moved by the algorithm should be explicitly clamped down, by specifying the target points for those points as the input points themselves. If, however, the algorithm does not deform the mesh to the user's satisfaction, the user can interactively refine the mesh deformation by manually adjusting the mesh points.

Note that if the lattice is considerably coarse as compared to the mesh, its effect is not localized to a very small area. A modified version of the algorithm uses a radial decay function (this feature can be turned on/off by the the user) that localizes the effect of the moved mesh vertices to smaller regions around them rather than to all the vertices inside the affected lattice cell. In such a case the movement of a mesh vertex is damped by a weight w, which is calculated as follows: Let r_{max} be the maximum radius of influence of the radial decay function (this is a user-defined quantity). Let mv be the input mesh vertex being moved and mv' be any other mesh vertex, then a weight w is calculated as

$$w = \begin{cases} 0 & \text{if } d \geq r_{max} , \\ 1 & \text{if } d = 0 , \\ \left[1.0 - \left(\frac{d}{r_{max}}\right)^2\right]^2 & \text{if } d < r_{max} , \end{cases} \tag{3.11}$$

where d is the Euclidean distance between mv and mv'. A point mv' may get assigned multiple weights when more than one mesh point is moved. In such a case only the maximum of all those weights is finally retained. Hence, if a mesh point lies outside the radial region of influence of the input mesh point, it does not move at all. The barycentric parameterization of the mesh vertices changes when the radial decay function is used and has to be recomputed every time, but because it is a linear parameterization, recalculation is not costly. The layering of the deformation engine allows the user to look at and manipulate suitably the skeletal and nonskeletal deformation separately without causing unwanted artifacts in the mesh (as is also observed in [91], who also match the silhouette but use curve matching in 2D to do it). Though an adaptive hierarchical lattice, which can be subdivided finely in areas where more control over the movement of vertices is required, will be a better solution for the localized deformation problem than the radial decay function, we use the decay function because it is easier to implement. The radial decay function is used only to localize the effect of the DFFD and not for the actual deformations themselves as the least-squares solution computed by the DFFD is much more efficient. Also, the DFFD provides higher level control over groups of vertices due to the presence of the lattice, while manipulating individual vertex deformations using a radial decay function is much more cumbersome.

Alternatives to DFFD, like the work on sketching mesh deformations by Kho and Garland [78], can be used as the deformation model in the framework. It uses curves sketched on the mesh model to directly deform the mesh. This will also improve the performance of the algorithm as it will give finer level control over the movement of the mesh vertices. The view-dependent deformation algorithm, however, does not change.

The mesh obtained after this stage is the *deformed mesh model* [see Fig. 3.5(e)]. This process of camera recovery, posing, and mesh deformation is repeated for each of the character sketches, and we get a camera and a character pose pair corresponding to each sketch. We refer to these cameras as *key views* and the associated character poses as *key poses*. These are used for creating the view-dependent animation. It is worthwhile to note here that computer vision-guided character posing and deformation require significantly less work than is required in doing the same manually.

3.6 Animating the Character

After we recover the view-dependent model for every sketch provided by the animator, we construct the view space. We illustrate the technique with the help of an example.

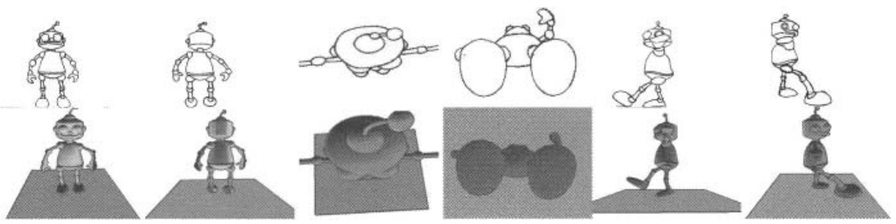


Fig. 3.12. Hugo sketched from six directions, namely front, back, top, bottom, left, and right. The second row shows the corresponding animation key frames generated by the system. Notice the marked perspective foreshortening in the top and bottom view sketches and how the effect is correctly reproduced by the recovered camera. In the last sketch Hugo's right leg undergoes a marked mesh deformation, which cannot just be obtained by skeletal pose recovery. This is as per the corresponding sketched view.

3.6.1 Constructing the view space

An animator provided the set of sketches shown in the top row of Fig. 3.12. the framework recovered the poses along with their corresponding cameras, as shown in the bottom row of Fig. 3.12. In this example the ability of the framework to recover and use the full projective camera is seen clearly in the top and bottom views where the projective foreshortening effect is very pronounced. The 3×4 projective camera matrix, **P**, is decomposable as **K**[**R**|**t**], where **K** is a 3×3 matrix containing the focal length of the camera, **R** is a 3×3 submatrix controlling the view direction, and **t** is a 3×1 submatrix governing the viewpoint distance (see Section A.1). Hence, we can recover all this information from the camera matrix. We recover the view direction v as

$$v = det(\mathbf{M})\mathbf{m}^3 , \tag{3.12}$$

where **M** is the first 3×3 submatrix of **P** and $\mathbf{m}^{3^\top}$ is the third row of **M**. $det(\mathbf{M})$ is the determinant of **M**. The camera center **C** can be estimated as the right null space of **P** by solving $\mathbf{PC} = 0$. The look-at point, l, is given by

$$\mathrm{l} = \mathbf{C} + \lambda v . \tag{3.13}$$

We normalize these view directions to get the key view directions, v, for the view space. For this example, all the look-at points coincide; hence the view space is reduced to a sphere (see Section 2.2). In Fig. 3.13, the large sphere is the view space, the smaller red spheres represent the camera centers, and the red lines represent the view directions.

3.6.2 Generating the animation

To generate the animation we need to compute the novel views for all points p on the camera trajectory $P(\underline{v}, \underline{t})$. We first find the r-closest key viewpoints by using a radial distance based selection, i.e., a key viewpoint v lies in the r-closest set of a point p if

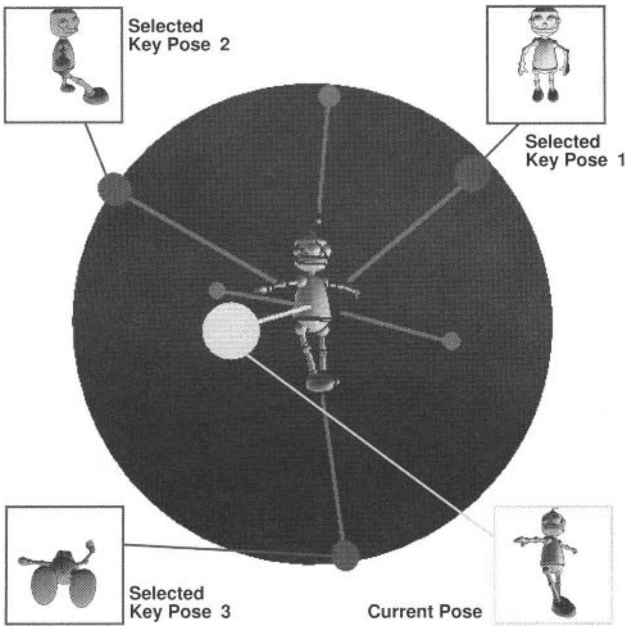


Fig. 3.13. The view space — the smaller blue and red spheres are the key viewpoints (see colour insert).

$d(\underline{v}, v) \leq d_{thresh}$. Here, $d(\underline{v}, v)$ is the distance between $\underline{v}$ and v measured on the envelope. The distance measurement is dependent on the coordinate system in which the view space is defined. It can be made independent of scale if the coordinate system is normalized. d_{thresh} is the distance threshold decided by the animator depending on the density and position of the views available and the intended animation. For dense views and if d_{thresh} is small, the distance on the envelope can be approximated by a chordal distance. Once the r-closest set, i.e., the $\bar{v}$'s, have been determined, we compute the blending weights as follows:

$$w_{\bar{v}} = \left(\frac{1}{d(\underline{v}, \bar{v})}\right)^{\alpha}, \tag{3.14}$$

where $\bar{v}$ are the selected r-closest key viewpoints and $\alpha \geq 1.0$. Suitable values of α generally vary from 2 to 4, again depending on the density and position of key views. The blending weights are normalized such that $\sum w_{\bar{v}} = 1$. For numerical stability, when any $d(\underline{v}, \bar{v}) < \epsilon$, the corresponding $w_{\bar{v}}$ is clamped to 1 and all others are set to 0. Here ϵ is a very small number like 1×10^{-6}. This ensures that the pose matches the key pose exactly when the current viewpoint is at a key viewpoint. It also assumes that the key viewpoints are more than ϵ distance apart so that there is no ambiguity in selecting a key viewpoint due to the clamping. This is a valid assumption because ϵ is very small. The resulting blended pose at p is calculated using Equation (2.1).

Figure 3.13 shows such a blending using the radial blending function. The blue spheres are the selected r-closest key views (the $\bar{v}$'s) for the given current viewpoint (shown as the green sphere). The current pose is a blend of the corresponding selected key poses. Note that the weighting function used here is an instance of the more general framework. Other functions can also be used to select the r-closest poses and for calculating the blending weights. It is possible to give the animator a choice between various weighting and blending functions. The value of α is also user controlled.

A smooth camera path guarantees a smooth animation because the blending function used to compute the pose at any point on the path ensures that the in-between poses are a smooth blend between the selected key poses. The blending function is continuous over its domain. We also assume that the pose specified by the animator for any key viewpoint is similar to the pose specified for any other key viewpoint in its small neighbourhood. Thus, the animation does not have any sudden unintended changes in the view or pose between successive frames.

Dense or uniform sampling of views in the view space is not a requirement of the method. The animator is free to populate the view space as she wishes, in order to get the desired animation. A pose can be added to an existing view space. We have an example of such an augmentation in Section 2.3, where we add two new poses to the view space created for the *Hugo's High Jump* animation to get an animation changing with changes in distance of the viewpoint.

3.6.3 Blending view-dependent animation with non-view-dependent animation

Let us examine what we mean by non-view-dependent animation. Any animation where the character's action is not explicitly dependent on the camera movement is non-view-dependent. Such animations may even have stationary cameras. Hence, any animation generated using normal keyframing will fall into this category. In order to fit into a conventional animation pipeline, the framework has to be able to blend a view-dependent animation sequence with a non-view-dependent animation sequence seamlessly, in terms of the camera shot.

We know the cameras for the last few frames of the first sequence. One idea is to maintain the same camera in the second sequence if it is a static camera sequence. If the camera in the second sequence is positioned differently, then we transition toward the camera in the second sequence gradually, interpolating the camera positions across the seam using a spline curve.

We use the *Hugo's High Jump* animation as an example in Chapter 2 to explain the view space creation. In this animation, we have generated the actual jump as a view-dependent animation (that part is generated from the poses given in Figures 2.7 and 2.16). Hugo's run-up before the jump, however, is generated using simple keyframed animation, and it blends in seamlessly with the view-dependent portion. Here the camera simply translates with the character during the run-up and transitions into the view-dependent camera when Hugo starts the jump.

3.7 Discussion of Other Results

We have generated various view-dependent animations using sketches as input, using techniques we have so far discussed. Here we present the salient features of the framework demonstrated by each of these animation results.

We have already discussed the *Hugo's Antics* animation sequence (the input sketches and corresponding rendered key frames are shown in Fig. 3.12). This sequence is a concept demonstration. The character used in the sequence is called Hugo. Some of the sketches have a marked perspective foreshortening effect, and hence camera recovery warrants the use of a full projective camera recovery. Further, in each of the sketches Hugo is posed and deformed differently. While some of these deformations can be obtained directly by reconstructing the skeleton pose, at other places the mesh needs to be deformed to match the sketched character more accurately. The animation sequence has the camera going around Hugo, cycling through each of the key viewpoints with brief pauses at the top and bottom key viewpoints to highlight the correctness of the full projective camera recovery. Hugo deforms accordingly in response to the camera. When the camera is to the right of Hugo, his right leg undergoes a marked mesh deformation that cannot be obtained by just skeletal pose recovery. This is as per the corresponding sketched view. Other poses have less mesh deformations.

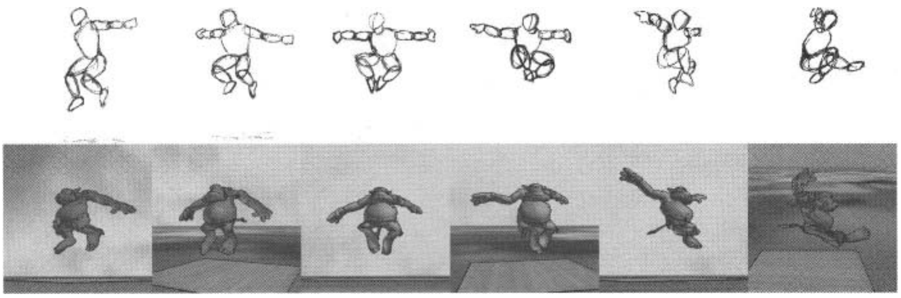


Fig. 3.14. Sketches and corresponding rendered key frames from the *Olaf Reloaded* animation.

The *Olaf Reloaded* clip is inspired by the opening *freeze frame – camera rotate* shot from the movie *Matrix* [131], filmed on the character called Trinity. The character used in the animation is the Olaf, the Ogre. As the camera goes round Olaf, he replicates the midair kick made famous by the movie. This sequence has been generated using mannequin sketches. This demonstrates that view-dependent animation is indeed possible using such sketches and the skeletal pose reconstructed by the system. The camera model used in this sequence is affine. We stress here that all the changes in Olaf's pose is in response to the camera movement, and it should not be confused with a rigid rotation of the character. This animation required six sketches for the view-dependent sequence (see Fig. 3.14).

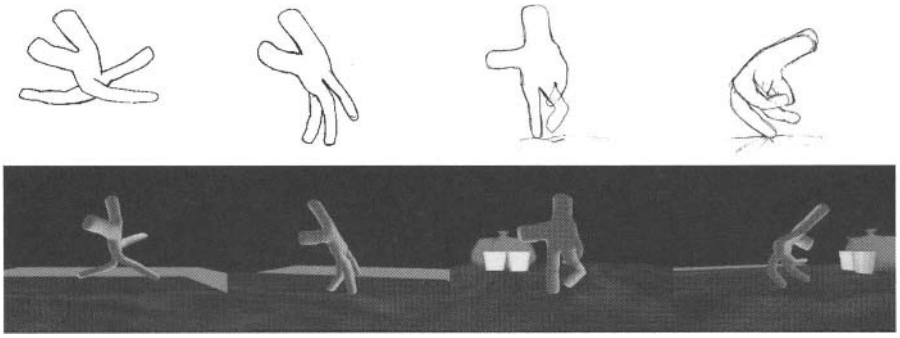


Fig. 3.15. Sketches and corresponding rendered keyframes from the *Ballet of the Hand* animation.

The longest animation clip we have generated using this technique is titled *Ballet of the Hand*. The character used is a cartoon hand (and hence the four fingers). Affine cameras are used in this animation. This clip has three view-dependent sequences. The first has the hand trying to do a jump very characteristic of a ballet sequence. Notice that the legs (of the character) do not stretch far enough to execute a perfect ballet jump the first time, so the character retries the jump. The retry attempt of the character has been generated by changing the camera angle and replaying the same sequence again. It can be clearly seen that during the jump the legs stretch further apart this time, and hence we clearly demonstrate deformation changing in response to a changing camera. As a final demonstration of the elegance of the view-dependent animation technique the character executes a *pirouette* (a spin move in ballet), and it can be seen as the camera also rotates independently of the character; the character clearly deforms *while rotating*, in response to the camera movement. All the three view-dependent sequences together required less than 10 sketches as input. We give some of the sketches and the corresponding rendered keyframes from the animation in Fig. 3.15.

The *Hugo's High Jump* animation sequence we discussed in Chapter 2 is also generated from sketches. The various sketches used for inputs are shown in Fig. 2.7, while the recovered cameras are shown in Fig. 2.8. The generated view space with a sample camera path is shown in Fig. 2.9. The complete animation starts with keyframed animation sequence where Hugo runs in and prepares to jump. In this part of the animation the camera merely translates with the character. As Hugo begins his jump, the camera begins to go around him and the sequence seamlessly blends into a view-dependent animation sequence. As Hugo completes his high jump and falls, the sequence again blends back into a keyframed sequence. We also demonstrate a variation of this animation by tracing a different path on the view space. On this path the distance of the viewpoint from the character changes, and the animation responds to these changes as explained in Section 2.3. This also illustrates that there exist other paths on the view space (other than the path planned prior to the creation of the view space) that are capable of generating interesting animations.

3.8 Chapter Summary

In this chapter, we have presented a complete pipeline to create a view-dependent animation from sketches. A review of the prior work shows that sketch-based modeling has been a popular technique for creating animations. However, there have been relatively few attempts toward posing characters from sketches. These techniques use simple camera projection models like orthographic cameras and rely on the user to disambiguate between possible 3D character poses. Research has also been done to deform a partly posed character mesh so that it exactly matches a sketched pose.

The pipeline to create a view space from sketches takes as input a set of sketches and a base 3D mesh model of the character. We use robust computer vision algorithms to recover the camera that best aligns the 3D character to the view direction in the sketch. Then we determine the pose of the 3D character that best matches the sketched character pose when seen through the recovered camera.

We present two novel view-dependent algorithms to ascertain the pose of the 3D character. The view-dependent posing algorithm poses the character by moving the skeleton embedded inside the character (using IK) such that the pose matches the sketched pose when viewed through the recovered camera. The view-dependent deformation algorithm moves the vertices of the 3D mesh (using DFFD) in order to ensure that the silhouette curves of the 3D mesh project onto the corresponding curves of the sketch when projected using the recovered camera. In this way we are able to embed a multilayered deformation system (IK and DFFD) into a view-dependent setting by using them in conjunction with a recovered camera estimated with computer vision techniques.

Each recovered camera and character pose pair forms a view-dependent model. We then show how the view space is formed from these view-dependent models recovered from the sketches. The animation is generated by tracing a path on the envelope of the view space. We also present many example animations of varying complexity, generated using this pipeline, to validate the claims about the effectiveness of the techniques presented in this book.

We have chosen to use IK and DFFD and have modified and integrated them into the framework. We, however, would like to point out that other viable alternatives to these techniques exist in recent literature (see Sections 3.5.1 and 3.5.4). These can be suitably adapted into forms that can replace IK and DFFD without affecting the view-dependent posing and deformation algorithms. The implementation of the pipeline that we present in this book should be considered as a concept demonstration.

In the next chapter we explore how we can use a combination of multimodal inputs like video and sketches to enhance the scope of the framework and create more interesting view-dependent animations.

4

View-Dependent Animation from Multimodal Inputs

In this chapter we present techniques for generating view-dependent animations from multimodal inputs. We have already seen in Chapter 2 the theoretical framework used to represent view-dependent animations. In the preceding chapter, we presented the complete pipeline to generate moving-camera character animations from sketches. Animators, however, use many different kinds of inputs to create an animation, such as video and motion capture. We wish to harness the various input methods, either separately or in combination, to generate better view-dependent animations.

First we examine and address the challenges associated with using multimodal inputs for creating moving-camera character animations. We start from video-based animation. Video is different from a set of sketches primarily because there is a temporal component to the video. We study the use of video for creating animation in the existing literature. We then explain how the information contained in a video is mapped to a view space to generate a view-dependent animation. We present the pipeline developed for this purpose. We then generalize this solution for video to incorporate other forms of input. We show that the framework can handle any combination of the various input methods and it offers considerable freedom to the animator in creating moving-camera character animations.

4.1 Challenges in Multimodal Authoring of Animation

Authoring moving-camera character animations from multimodal inputs is a challenging task. The main problems we face while working with multimodal inputs are:

1. The primary components of a view-dependent animation are the key cameras and the key character poses associated with them. In order to create view-dependent animations, we must be able to extract information about these components from the various types of inputs we want to handle.
   Different types of inputs like sketches, video, and motion capture have different characteristics and, hence, require different techniques for extracting these components. A sketch is a single snapshot of the character's pose from some

camera, and in Chapter 3 we presented the techniques to infer this information from sketches. A video is a sequence of snapshots taken over a period of time. Since the camera and the character pose may change continuously throughout the video, these have to be tracked across time. Both camera and character tracking are very challenging problems. Motion capture is yet another type of input, which makes use of multiple, static, calibrated cameras. The character data output from motion capture systems is usually in the form of time-varying joint state vectors. As a first step, therefore, we have to develop suitable techniques to extract the required information from these inputs, to create the view-dependent animation.

2. Next, we wish to utilize all the information, extracted from these inputs to create the desired animation. In order to do this we need to combine the information in some meaningful manner. The challenge here is that the camera and character pose information extracted from various sources may have different representations. For example, information extracted from a sketch may have the camera represented as a camera matrix and the character pose recovered as a skeleton. In a video, however, the camera position information is obtained as a stream of continuously tracked rotations and translations. The character in a video may be tracked in many ways and the tracked information may include contour positions, blob positions, or 2D skeleton configurations. Similarly, motion capture systems may have other representations for the captured motion information. It is extremely tedious and confusing to work with numerous, different abstractions and formats. We must be able to interpret them and map them to a common representation. This will allow us to combine them effectively and create the desired animation easily.
3. The number of cameras and character poses extracted from the various inputs can turn out to be quite overwhelming, even after they are all mapped to a common representation. Therefore, we must provide the animator with an interface to explore them. The animator can create the view-dependent animation easily and efficiently if provided with a tool for interactive visualization of this collection of cameras and character poses and see the resulting animation in real time.

In subsequent sections, we present solutions to all these problems. We first examine the prior work done on creating animations from video and then present a technique for generating view-dependent animations from video.

4.2 Prior Work

The use of video for generating animations has been extensively researched. In spite of many years of active research, capturing motion from video remains a very challenging problem. As we have already mentioned, a video is different from a sketch because it is a sequence of snapshots over time. Since we want to use video to create view-dependent animations, we must be able to extract the character pose and the camera parameters from it. Thus, we look at existing work in this area under the two broad categories of character and camera tracking in video.

4.2.1 Character tracking in video

Character tracking is more commonly known as *motion capture*, because tracking the character is equivalent to recording of the motion of the character using different sensors. Even after substantial progress in video-based motion capture techniques, animation from video remains a hard problem. Gleicher and Ferrier [44] describe why the demands of animation pose a difficult challenge for video-based motion capture techniques. Animation is extremely sensitive to small jitters and wobbles which are always present due to noisy capture and inaccurate computations. Moreover, the importance of high frequencies in capturing subtle nuances of a motion implies that naive filtering is not a viable tool for noise removal at video sampling rates. Also, the unpredictability and unusual motions that need to be captured limit the strength of the models that can be applied.

In subsequent sections, we examine the various techniques that are used to track human and other characters in video for generating animation.

Tracking human characters in video

Animating humans from video performances is one of the primary objectives of video-based motion capture systems. These systems use computer vision techniques to analyze the video data and interpret the motion of the character from it. Moeslund and Granum [102] present a comprehensive study of computer vision-based human motion capture literature from the past two decades. A general computer vision-based motion capture system can be functionally structured into four stages. Before a system is ready to process data, it needs to be *initialized*; i.e., an appropriate model of the subject has to be established. Next, the motion of the subject is *tracked*. Tracking includes segmenting the subject from the background and finding correspondences between segments in consecutive frames. The pose of the subject's body often needs to be estimated. Many times this itself may be the output of the system, for example, to control an avatar (the graphical representation of a character) in a virtual environment the body pose information is required. High-level knowledge, like a kinematic model, is typically used in *pose estimation*. Depending on the requirements of the animation, the pose information may be further processed in order to *recognize* the actions performed by the subject.

Model initialization has two aspects: to find the initial pose of a subject and to define the model representing the subject. In some systems this problem is reduced either by assuming the subject's initial pose to be known as a special start pose [27] or by having the operator of the system specify it [141]. Perales and Torres [106] specify it for every frame, while Zheng and Suezaki [144] manually fit the pose at some key frames and then use correlation to interpolate between frames.

The primary purpose of tracking is to extract specific image information, either low level, such as edges [61], or high level, such as hands and head [139] positions of the subject. Independent of the context, three common aspects of tracking can be identified. First, the tracking algorithm needs to separate the moving character from the rest of the image, i.e., the figure-ground problem. When the background and the camera are relatively static, temporal data like background subtraction [5] or flow [20, 141] can be used for this purpose. Spatial-data-based techniques use thresholding [69] or statistical approaches like deformable templates [16] or hidden Markov models [110]. Second, these segmented images are transformed into another representation to reduce the amount of information or to suit a particular algorithm. The third aspect defines how the subject should be tracked from frame to frame. The correspondence analysis is often supported by prediction. A model of velocity and acceleration or more advanced models of movements such as walking [111] may be used. A commonly used method for prediction is the Kalman filter [73], suitable for unimodal distributions. The CONDENSATION algorithm [68] is capable of tracking multiple hypotheses, i.e., support multimodal distributions, and has been shown to be a powerful alternative to the Kalman filter. A prediction-correction framework may sometimes fail for complex cases. Zheng and Takagi [145] provide a graphical interface to manually correct predicted model tracks.

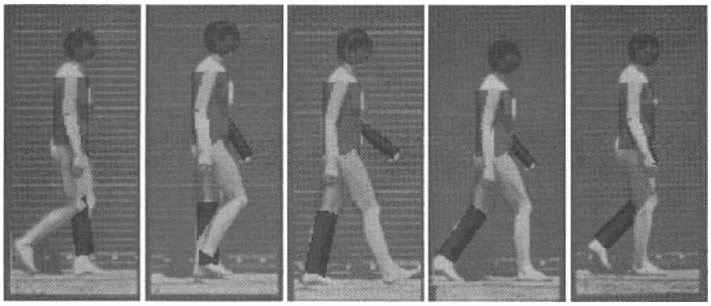


Fig. 4.1. Tracking results on Muybrige's Woman Walking [103]. The reconstructed pose is projected on the input frames (images courtesy Bregler and Malik [23]).

Pose estimation is the process of identifying how the character's body and/or individual limbs are configured in a given scene (see Fig. 4.1). Often, explicit kinematic and dynamic models of an articulated body are used for this purpose. The concrete representation of the character's model is a state space where each axis represents a degree of freedom of a joint in the model. A pose of the subject may be expressed as a point in the state space as opposed to many points in the 2D image space. The idea is to predict the pose of the model corresponding to the next image. The predicted model is then synthesized to a certain abstraction level for comparison with the image data. Constraints are introduced to prune the state space, either by partitioning the state space into legal and illegal regions, as in [101] or by defining them as forces acting on an unconstrained state space, as in [140]. Another approach to reduce the number of possible model poses is to assume a known motion pattern, especially cyclic motion, such as walking and running [111]. The various abstraction levels used for comparing image data and synthesized data are edges [60], silhouettes [75], contours [71, 72], sticks [87], joints [50], blobs [140], depth [108], texture [117], and motion [23]. The recognition aspect of motion capture can be seen as post processing and is not particularly relevant to the problem at hand.

Kakadiaris and Metaxas [72] present a system for animating customized virtual humans using motion parameters estimated from multiview image sequences. First, a subject is asked to perform a set of motions to acquire the anthropometric dimensions of his/her arms. The system is initialized manually by matching the initial configuration of the model to the starting frame, and then a Kalman filter is used to predict the state of the model in subsequent frames. Based on the predicted position of the model, the system synthesizes the appearance of the model as it would have been seen by the different cameras. For every model the projection of the vertices of the model to the image plane of each camera is computed, and a new state for the model is estimated that minimizes these discrepancies. Collomosse et al. [30] generate a camera-motion-compensated version of the sequence, thereby ensuring that camera motion does not influence the observed trajectories. They track features over the sequence and analyze the occlusion of features during tracking to determine their relative depth ordering.

Tracking other characters in video

Animation deals with a wide spectrum of characters (see Chapter 1). Techniques for tracking based on assumptions specific to human motion or the human body may not work for other than human characters. Efforts have also been made to capture the expressive movement styles usually found in traditional animation (see Fig. 4.2).

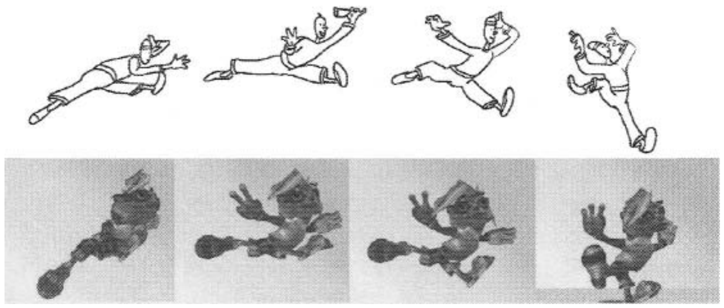


Fig. 4.2. Motion of a jumping character retargeted to a 3D model (images courtesy Bregler et al. [22]).

Bregler et al. [22] describe a method to capture the motion from a cartoon animation. They parameterize the motion with a combination of affine transformations and key weight vectors. The affine transformations encode the global translation, rotation, scaling, and shear factors. Key shape deformations, which are defined relative to a set of key shapes, are used to capture finer deformations. Shapes between the key shapes are approximated by a multiway linear interpolation. The cartoon shape is obtained from video by contour tracking. The user defines for each input key shape a corresponding output key shape in 2D or 3D. Then the estimated motion parameters can be mapped from the 2D source to the 2D or 3D target. This strategy of designing input-output key shape pairs circumvents many problems that arise in standard skeleton based motion adaption. This work attempts to bridge the gap between techniques that target the traditional expressive animation world and the motion capture based animation world. Motion capture data is appropriate for domains that require realism, whereas, traditional animation data is better suited for areas that call for more expressive and stylistic motion. This paper shows that traditional animation footage can be treated like motion capture data.

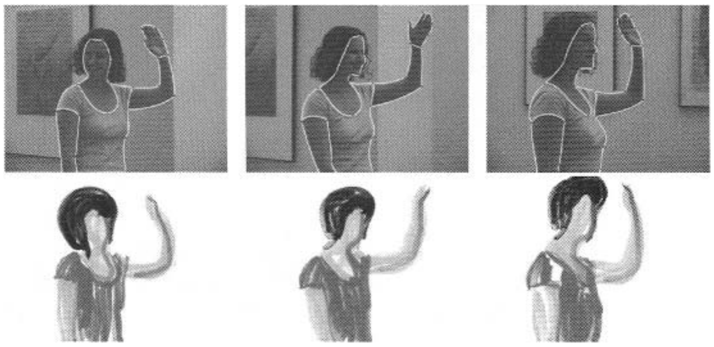


Fig. 4.3. The top row shows the input frames with the tracked *roto* curves. The bottom row shows an animation that follows the curves. An artist draws all strokes in the first frame; the later frames are generated automatically (image courtesy Agarwala et al. [3]).

Agarwala [2] presents a rotoscoping system that also uses contour tracking to generate animations from video (see Fig. 4.3). The user sketches contours directly onto the first frame of video. These sketches initialize a set of spline-based active contours that are relaxed to best fit the image. The system then uses motion estimation techniques to track these contours through the image sequence. The contours are then used directly as primitives for the animation. This work is further improved upon by Agarwala et al. [3] where they describe a contour tracking algorithm cast as a space-time optimization problem to do rotoscoping. The user specifies the positions of the *roto* curves in the starting and the ending key frames. Then the system determines the positions and shapes of these curves in the intermediate frames by solving a space-time optimization problem. The objective function is a linear combination of five weighted energy terms depending on the change in length of the vector between adjacent samples on the curve, change in curvature, rate of motion, flow along normals to the curve points, and image gradient at points along the curve. The user may edit the resulting sequence to correct the errors produced by the optimization. These are then used to create 2D cartoon-style animations from video.

In another contemporary work by Wang et al. [132], video input is semiautomatically rotoscoped into semantically meaningful regions using a mean shift-guided interpolation algorithm. These are then used to generate 2D cartoon animations.

Video-based rendering of character animation

In contrast to the above works, video-based techniques such as video interpolation [21, 34] and video sprites [116] provide an image-based representation of dynamic scenes allowing synthesis of novel image sequences with visual quality comparable to the captured video. Starck et al. [120] present a system for video-based representation for free viewpoint visualization and motion control of 3D character models created from multiple-view video sequences of real people. They represent character motions as free viewpoint video reconstructed from multiple-view video sequences captured in a studio with 10 cameras. The motion sequences are blended to provide seamless motion cycles and smooth transitions between different motions. The blended video sequences are constructed as an offline process, and a motion graph is used to represent them and to create a 3D character animation.

4.2.2 Camera tracking in video

Matchmoving or camera matchmoving refers to the process of matching the position and angle of computer generated synthetic objects (such as an animated character) to real footage shot with a film or video camera. Graphical objects should be inserted such that they appear to move as if they were a part of the real scene. Seamless, convincing insertion of graphical objects calls for accurate 3D camera motion tracking, stable enough over extended sequences so as to avoid the problems of jitter and drift in the location and appearance of objects with respect to the real scene. Matchmoving finds several important applications in animation, augmented reality as well as in the creation of special effects [135]. In order to provide the versatility required by such applications, very demanding camera tracking requirements, in terms of both accuracy and speed, are imposed [11].

Camera tracking algorithms are well documented in recent computer vision literature [41, 118]. They generally involve the following steps:

1. Compute feature points in each video frame (feature detection).
2. Match the feature points between neighbouring frames (correspondence).
3. Compute a projective reconstruction from the interest point matches (compute feature tracks).
4. Compute a metric reconstruction (autocalibration of the camera).

Camera tracking methods often need to work in unprepared, unstructured scenes, large-scale environments, or archive footage. Methods that avoid making any assumptions regarding the environment exploit geometric constraints that arise from the automatic extraction and matching of appropriate 2D image features such as corner points. Corners are simply points of localized image structure, formed at the boundaries of image regions of different brightness. Depending on their mode of operation, proposed approaches can be classified into two categories.

The first category consists of methods designed for offline use on prerecorded image sequences [32, 40]. Such methods process image data in a batch mode and usually are noncausal, employing both past and future frames for deducing the camera motion corresponding to the current frame. Albeit accurate, batch techniques are computationally demanding due to the use of global bundle adjustment, which involves the solution of large, nonlinear optimization problems [128]. This, plus the requirement of operating on the whole sequence at once, makes batch methods inappropriate for use in time-critical applications.

Methods operating in a continuous mode, in which images are processed incrementally as acquired, constitute the second class of camera tracking techniques [10, 14, 118]. Typically, such methods are causal, relying only on past frames for estimating the camera motion for the current one.

A lot of work has been done in the areas of tracking characters and cameras from video. These two areas are, however, always treated separately. There is no single method that combines both. To generate a view-dependent animation, both the camera and the character need to be tracked. We use existing methods to track the camera (see Section 4.3.1) and the character (see Section 4.3.2), and then use the tracked cameras and character poses to generate the view space. This allows us to combine the camera and the character information extracted from the video into one representation. We have primarily focused on monocular or single camera video and, hence, do not require any special setup to work with.

Our tracking framework is initialized in the first frame by drawing the spline contours on the image. We also associate a 2D stick skeleton with the contour curves in the first frame (see Section 4.3.2). We use a Kalman-filter-based active contour framework, inspired by [16]. We find that it is suitable for tracking both animated characters and real people; so it can handle a variety of uses. In case the tracker fails for certain hard-to-track videos, e.g., for videos with prolonged self-occlusion of body parts, we provide interactive adjustments and restart the tracker. The system (see Section 4.3.2) based on [16] has many of the good features of the space-time optimization based system of Agarwala et al. [3]. The contour shape tracked can be biased toward an average deformable shape template to track rigid motions. The tracker can also be made almost independent of the average shape to track nonrigid motions and can even track agile motions in the presence of background clutter. It works at real time frame rates for tracking, which a system based on space-time optimization cannot do.

In the framework, for every video frame on which the contour is tracked, the 2D stick skeleton also gets tracked along with the contours. We then use the view-dependent posing algorithm (see Section 3.5.2) to pose the 3D skeleton of the character from the tracked 2D skeleton. We have used a video clip, with significant character and camera movement, as input to the pipeline for creating a view-dependent animation from video (see Section 4.3). Moreover, the framework also has the ability to incorporate stylizations from multimodal inputs such as sketches and videos.

In the next section, we present our pipeline to create the view space from video. We then argue that the pipeline can be easily extended to other forms of input. We show that if such a pipeline can be created, the information required to create the view-dependent animation can be extracted. We also show that the view space is an apt representation for this information.

4.3 Creating a View Space from Video

A video differs from a set of sketches since there exists a strong temporal correlation between the frames of the video. In order to take advantage of this fact, a separate set of processing techniques is required. However, we maintain the higher level structure of the animation pipeline developed for dealing with sketches so that the animator finds it intuitive and easy to use (see Section 3.2).

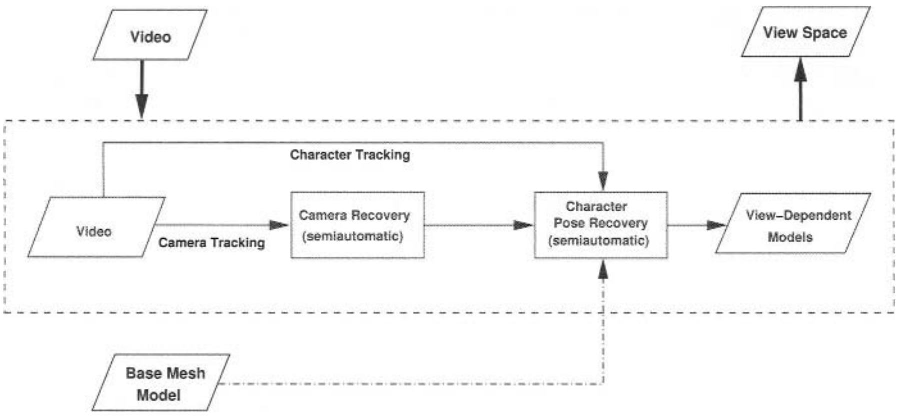


Fig. 4.4. Schematic diagram depicting the pipeline to create a view space from a video.

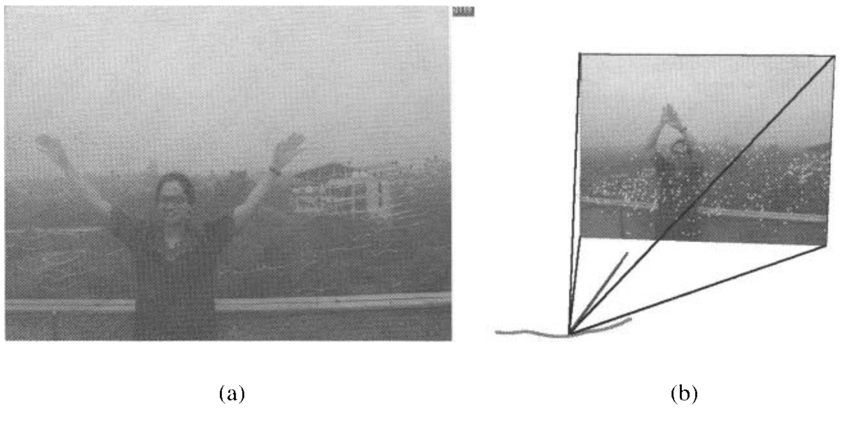
The input here is a video sequence of a character performing some action. In addition we require the base mesh of the character, which is to be used in the new animation. We also require the skeleton and the lattice as mentioned for sketches (see Section 3.3). We illustrate our method by using an example video sequence, which has significant camera movement during the character's action. We use this video as input to generate our view space and consequently a view-dependent animation. In order to recover the camera and the respective character poses, we track the camera and the character across the video. These are required to construct the view space from the video (see Fig. 4.4).

4.3.1 Camera tracking

The first stage in the pipeline shown in Fig. 4.4 is the recovery of the camera. In a video, however, the camera moves on a continuous path, except across shot changes.

Hence, we need to recover the camera position for every frame of the video. The solution is to track the camera across the video. For this purpose, we have used *Boujou* [18], a matchmoving software, which robustly implements the camera tracking algorithm. It processes video data in batch mode and is very accurate (see Section 4.2.2). The output of the camera tracking phase is the camera view direction and viewpoint position for every frame of the video. When the character mesh is viewed using these recovered cameras, it appears aligned to the view in the corresponding video frame.

In order to track the camera, we need to track the feature points on the background (i.e., the static part of the video). For this purpose, we mask out the moving character in all the frames of the sequence using an approximate polygonal mask. The masking out of the character has to be done manually only on the first frame, the last frame, and a couple of other key frames where the pose of the character differs significantly from neighbouring frames. The mask is then tracked and interpolated by Boujou for the whole video. We then track the camera from a dense set of background feature points (see Fig. 4.5(a)). Boujou performs camera tracking and returns the full projective camera matrix for each frame of the sequence. Figure 4.5(b) shows the camera path recovered (in red). We later use these tracked cameras to form our view space. The major advantage of the framework is that we can represent all the information returned in the full projective camera in terms of the viewing direction and the distance of the viewpoint from the character in the view space. Now we need to associate the character's pose with the cameras we recover.



(a) (b)

Fig. 4.5. (**a**) feature points tracked by Boujou, (**b**) camera tracking by Boujou. The camera path recovered is shown in *red* (see colour insert).

4.3.2 Character posing

After the cameras for each video frame have been recovered, we move on to the next stage in the pipeline (see Fig. 4.4), which is recovery of the character pose. As the character moves, the character pose also changes across the frames of the video. Hence, we have to also track the character across the video. We want a tracker that works reasonably well for rigid as well as agile motion. It should work in the presence of background clutter and should be able to track real people and animated characters. We draw inspiration from many of the prior works done in video-based animation [2, 3, 22], and use a contour tracker. We have implemented a contour tracker based on the work on *Active Contours* by Blake and Isard [16].

The tracker estimates a contour shape, represented by a spline curve with control points $\mathbf{X}(t)$, $\mathbf{Y}(t)$ at any instant of time. The tracker is initialized in the first frame by drawing the spline contours on the image. This defines an average shape template whose control points are given by $\bar{\mathbf{X}}(t)$, $\bar{\mathbf{Y}}(t)$. A shape transformation from the template to the current shape is given by $\mathbf{Q}$. The transformation for the template shape itself, $\bar{\mathbf{Q}}$, is the identity transformation.

The tracker then associates a Kalman filter with each contour. Kalman filters comprise two steps: prediction and measurement assimilation. Prediction employs a second-order object dynamics model to extrapolate past motion from one video frame to the next. Second-order dynamics, written in discrete time, can be defined using a state vector $\mathcal{X}$ as

$$\mathcal{X}_n = \begin{pmatrix} \mathbf{Q}_{n-1} - \bar{\mathbf{Q}} \\ \mathbf{Q}_n - \bar{\mathbf{Q}} \end{pmatrix}. \tag{4.1}$$

Successive video frames are indexed $n = 1, 2, 3, \ldots$

The dynamics of the tracked object is now given by the following difference equation:

$$\mathcal{X}_{n+1} = A\mathcal{X}_n + \begin{pmatrix} \mathbf{0} \\ \mathbf{w}_n \end{pmatrix}. \tag{4.2}$$

Here, A is a matrix representing the deterministic part of the dynamics, while at each n, $\mathbf{w}_n$ is an independent, normally distributed vector of random variables. The covariance of $\mathbf{w}_n$ specifies the random part of the dynamics model. The assimilation stage blends measurements from a given frame with the latest available prediction of the current state, $\hat{\mathcal{X}}_n$, using the following equation:

$$\hat{\mathcal{X}}_{n+1} = A\hat{\mathcal{X}}_n + K\begin{pmatrix} \mathbf{Z}_{n+1} \\ \mathbf{0} \end{pmatrix}, \tag{4.3}$$

where $\hat{\mathcal{X}}_{n+1}$ is the next predicted state, K is the Kalman gain matrix, and $\mathbf{Z}$ is the state-space measurement vector. The actual measurements on the image are in the form of feature points detected by the tracker along the contour normals. Since the tracker uses only intensity information to compute features, it works on gray-scale images. The contour shape tracked can be biased toward the average shape template to track rigid motions. It can also be made almost independent of the average shape to track nonrigid motions and can even track agile motions in the presence of background clutter.

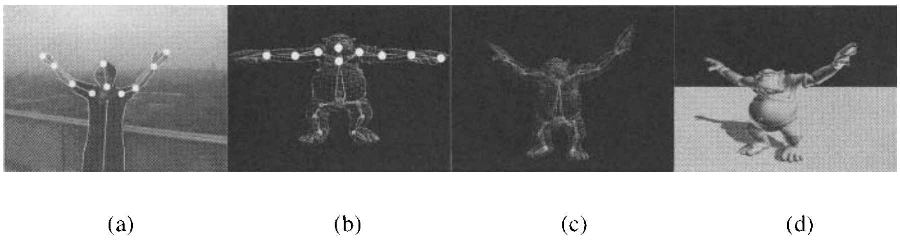


Fig. 4.6. Posing the character from a video frame: (**a**) contours tracked on a frame of the input video with joints of the 2D skeleton marked in *white*, (**b**) corresponding joints on the 3D skeleton marked in *white*, (**c**) 3D skeleton and character's mesh after posing, (**d**) final rendered pose of the character (see colour insert).

The example we show here is for an animated character, but a Kalman contour tracker is known to work for tracking real people as well. In case of prolonged self-occlusion of a body part — a situation no tracker can handle — we provide interactive adjustments and restart the tracker. Though a CONDENSATION tracker may behave better for kinematic singularities [39], we have used the Kalman contour tracker for its relative ease of implementation. We have found it sufficient for the purpose of demonstrating the concepts of this book.

We associate multiple contours with the character: one contour for every segment of the body, for example, arms, legs, hands, and feet. From the tracked contours we recover a 2D skeleton, whose segments represent the position of each body part. This is done by associating a 2D skeleton with the contour curves in the first frame of the sequence. We define the position of joints of the 2D skeleton relative to the points on the contour. Since this association holds across all the frames on which the contour is tracked, the 2D skeleton also gets tracked along with the contours. Thus, if the contour tracked is correct, then the 2D skeleton tracked is also correct. Figure 4.6(a) shows the tracked contours on one of the frames along with the associated 2D skeleton.

We pose a 3D character mesh using the tracked camera and the tracked 2D skeleton. In order to construct the view space, a set of cameras and corresponding character poses are required. The 2D skeleton is a good approximation of the character pose for an individual frame. However, a continuous sequence of these tracked 2D skeletons may have oscillations. In order to avoid this problem, the animator chooses a sparse set of appropriate frames from the video sequence. While selecting the frames, the animator also keeps in mind that these frames should be able to reproduce the motion from the video to a desired degree of accuracy. Posing is done only for these frames. In this example, the animator has identified 15 key frames for posing. Before the posing is carried out, the scaling of the camera coordinate system has to be matched with the scaling of the character coordinate system. This can be done if the estimate of a true dimension is known in the camera coordinate system. This allows us to set up an isotropic scaling matrix to be applied to the character before the posing process. The 3D character mesh has a skeleton embedded inside it [see Fig. 4.6(b)]. When this skeleton is moved, it moves the mesh along with it using blended vertex weights. The correspondence between the 2D skeleton and the 3D skeleton is specified only once during the whole process [Figures 4.6(a) and 4.6(b) show the joint correspondences marked in white]. The pose computed is such that the 3D skeleton projects onto the 2D skeleton for the corresponding camera. This is done using the view-dependent posing algorithm, given in Section 3.5.2. The resulting pose minimizes the error between the 2D skeleton and the projected 3D skeleton while maintaining kinematic constraints on the 3D skeleton. Figure 4.6(c) shows the posing of the 3D skeleton and the character mesh. The final rendered pose is shown in Fig. 4.6(d). The animator also has the choice of refining the poses manually, if so desired.

We track the contours on the image to obtain the 2D skeletons, but the contours themselves can be used to transfer mesh deformations by matching them to the silhouette curves of the mesh using the view-dependent mesh deformation algorithm (see Section 3.5.4). Semiautomatic techniques to do this have been demonstrated by [91] (see Section 3.1.2). Since in this example the character in the video and in 3D are not the same, manual intervention would be required on part of the animator in case such deformations are to be transferred meaningfully.

Figure 4.7 shows the posed character for two sample key frames. After we have posed the character for all the key frames chosen by the animator, we are ready to construct the view space.

4.3.3 Constructing the view space

We have the full 3×4 projective camera matrix, **P**, for every camera used for posing. Hence, as explained earlier in Section 3.6.1, we recover the view direction using Equation (3.12), and the look-at point using Equation (3.13). We normalize these view directions to get the key view directions, v. In Fig. 4.8, the red spheres represent the camera centers and the red lines show the respective view directions. The character in Fig. 4.8 is currently in its base pose. Note that all the view vectors do not intersect at a common point even though their general direction is toward the character. Hence, they do not lie on one sphere centered at a common look-at point, but in a general view space. Such a configuration of cameras cannot be handled by techniques like those in [109].

Since the animator has chosen 15 key frames from the video for posing, we use the corresponding v's from those frames to construct the view space. If we consider a camera path $P(\underline{v}, \underline{t})$ through these v, then at a time instant $\underline{t} : \underline{v} \equiv v$, the instantaneous view space is a unit sphere centered at the look-at end of the normalized view direction vector. This is because a unit sphere models the view space made of all the possible unit view direction vectors toward a particular point. The complete view space is the union of spheres centered at each of the look-at points (see Section 2.2).

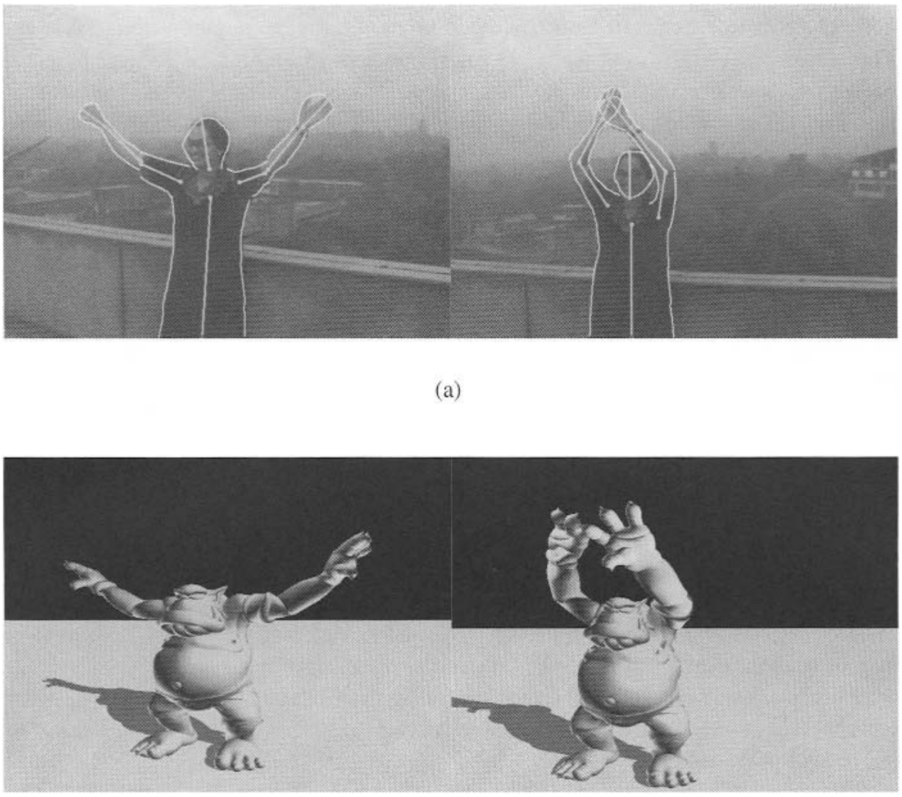


Fig. 4.7. (**a**) tracked contours and associated 2D skeletons on two key frames, (**b**) corresponding posed character viewed through their respective recovered cameras (see colour insert).

Figure 4.8 shows the view space generated (in green) for this example. Associated with every view direction, v, is the corresponding character pose, m_v. We incorporate the distance, d_v, associated with the corresponding view directions, v, by using the length of the vector $\mathbf{C}-\mathbf{l}$ [see Equation (3.13)]. Note that this view space construction is similar to the one explained in Section 2.2 for the *Hugo's High Jump* animation sequence. Hence, we see that the view space generated from sketch- and video-based inputs are constructed using similar pipelines. This makes the use of the framework extremely easy for the animator.

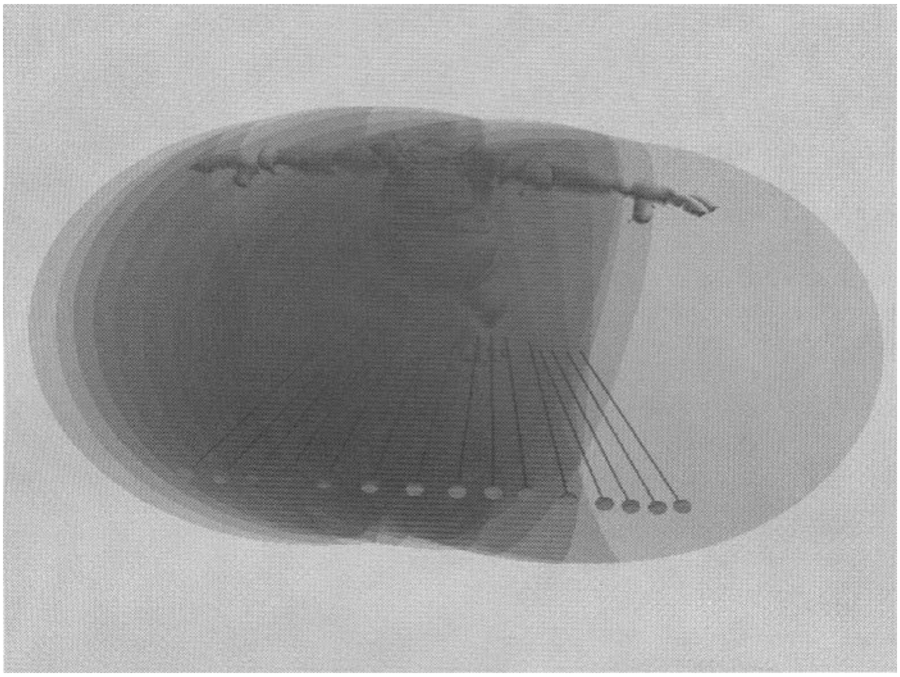


Fig. 4.8. The viewpoints, the view directions and the view space (see colour insert).

4.4 Creating the View Space from Multimodal Inputs

We can now map the information present in a set of sketches and a video to a view space. Even for other input methods, like motion capture, the basic blocks of the pipeline to create a view space will remain the same, i.e., the camera recovery stage followed by the character pose recovery stage. The view space can thus form a common representation for all these forms of input. It can be very easily used to fuse together these inputs. Thus we have solved the first two problems offered by the task of multimodal authoring. We have shown how to extract and meaningfully represent the information contained in multiple types of input.

The animator may choose to select some camera shots from a video, a few relevant poses from sketches, and motion data from motion capture and map them to the same view space. If synchronized video streams can capture a performance from multiple directions (as is done in a motion capture stage), then all these videos taken together can form a view space as they document the character poses from different directions over a period of time. This maps nicely into the framework and can be used to animate a new character easily. The animator can add more poses to vary the movement from the recorded performance, if so required.

In order to do all of the above, we must be able to recreate the camera and character motions from all our input streams. We must also be able to use a combination of various inputs as desired by the animator. In the next section we present a method to generate a view-dependent animation from the view space we create using our pipeline.

4.5 Generating the Animation from Video

In a video we have only one view direction at a given time instant, hence, all the camera shots recovered from the video maps to one particular path traced on the envelope of the view space. This path is the smooth path passing through all the recovered camera positions. In order to recreate the camera and the character motion from video, we move the camera along this path.

We need to compute the novel views for all points, p, on the camera trajectory, $P(\underline{v}, \underline{d}, \underline{t})$, to generate the animation. We summarize the process for generating the animation in Algorithm 4.1.

```
Require: The view space must be created before this algorithm can be run.
Require: The camera path $P(\underline{v}, \underline{d}, \underline{t})$ must be given.
1  begin
2      foreach point p on $P(\underline{v}, \underline{d}, \underline{t})$ do
3          Find the $r$-closest key viewpoints, $\bar{v}$, as explained in Section 3.6.2
4          $q_{\underline{v}} = \underline{d}\,\underline{v}$
5          foreach $\bar{v}$ in the r-closest set of $\underline{v}$ do
6              Compute the length of $q_{\underline{v}}$ projected along $\bar{v}$ as $\underline{d}\,\underline{v} \cdot \bar{v}$.
7              Find $d_{\bar{v}}^{l1}$ and $d_{\bar{v}}^{l2}$ such that $d_{\bar{v}}^{l1} \leq \underline{d}\,\underline{v} \cdot \bar{v} < d_{\bar{v}}^{l2}$, by a binary search in
               the sorted tuple list.
8              Compute $\beta$ using Equation (2.2).
9              Compute $w_{q_{\bar{v}}}$ using a radial blending function similar to the one
               given in Equation (3.14) with $d(q_{\underline{v}}, q_{\bar{v}})$ as the distance term.
10             Ensure that $\sum_{\bar{v}} w_{q_{\bar{v}}} = 1$.
11             Compute the blended character pose for $q_{\underline{v}}$ using Equations 2.3
               and 2.4.
12         end
13     end
14 end
```

Algorithm 4.1: Algorithm to generate a view-dependent animation from a view space.

Figure 4.9 shows blending using the radial blending function. Figures 4.9(a) and 4.9(b) show the character poses for two of the key views. The green sphere in the figures is the current viewpoint and the green line is the current view direction. The blue spheres are the r-closest key views for the current view position. When the green sphere is at a key view the pose matches exactly with the character pose associated with that key view even though other key views may be in the r-closest set. This is because the weight associated with that key view becomes 1 and 0 for the others. Figure 4.9(c) shows that when the current view (green sphere) lies between two (i.e., $r = 2$) key views (blue spheres), the character pose is a blend of the two poses.

For every point on the camera path, the algorithm requires a single pass through all the meshes that have to be blended (in the last step of the inner loop) and is, therefore, $O(n)$, where n is the total number of vertices of the meshes (i.e., it is linear in the number of vertices). The computations involved are quite simple, and the animation is generated in real time, as the user specifies a camera path, at frame rates varying from 30 to 60 frames per second (fps) depending on the density of views in the view space. The machine used is a Pentium 4 at 1.2 GHz with a GeForce 2 MX graphics card.

The algorithm generates the same camera and character motion captured from the video on a new character. We also have another example to demonstrate the generation of view-dependent animation from video, in which we use a short clip from DreamWorks Animation SKG's *Shrek* [1] to generate a view-dependent animation. In the input sequence, Shrek is flexing his muscles in the arena after having beaten-up the Lord Farquaad's guards, as the camera pans across the arena. This sequence was selected because it also has significant camera motion during the character's action. We cannot reproduce the original input due to copyright restrictions, but the generated view-dependent animation can be seen.

The extent of similarity of the movement in the new animation to the original movement in the video is a direct consequence of the number of sampled key views used to create the view space (15 in this example). A denser sampling will recreate the motion more faithfully than a sparse sampling but requires more work. Decision of the threshold for the sampling-density-versus-animation-quality trade-off is left to the animator. The algorithm, however, works well with a relatively sparse set of key views and associated poses also. The movement style of the generated animation can be altered during the posing phase in order to create a stylized variation of the movement recovered from the video. Another point to consider is that all the characteristics of the original video cannot be directly mapped to any new character. In the example where we have used the clip from *Shrek*, the original video has a lot of fine-level facial animation and some secondary cloth animation. This cannot be reproduced on the new character because we do not have requisite animation controls for extensive facial and cloth animation on the new character. The posing algorithm we use essentially reconstructs the poses at the sampled key frames. It does not explicitly make use of the temporal continuity information present between the key frames. This can be incorporated in the posing process by tying up the posing algorithm to the character tracking algorithm in an error-correcting feedback loop. This will further enhance the performance of the posing algorithm.

Note that in this particular example, associated with every key view direction, v, there exists exactly one character pose, m_v, at a particular distance, d_v. Hence, the tuple list associated with every v has length 1, and the blending of poses on the basis of distance along a view direction is not required. However, we illustrate blending of poses based on distance in another example explained in Section 4.6.

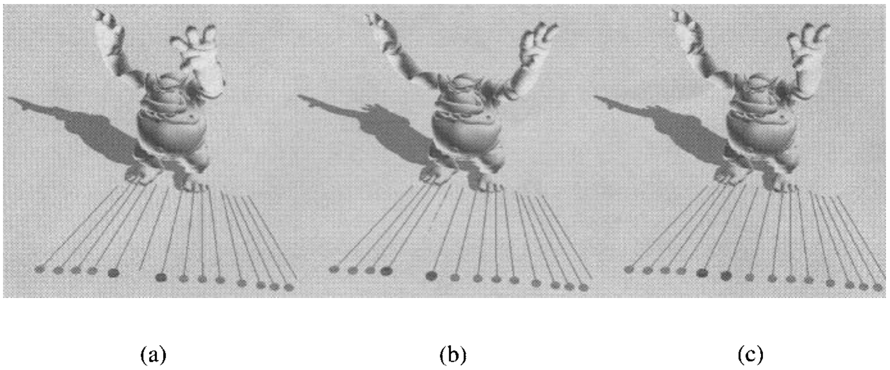


Fig. 4.9. Character poses associated with key views and novel view generation (see colour insert).

Figure 4.10 shows the camera path traced by $q_{\underline{v}}$ as a trail in green, at the correct distances as recovered from the camera tracking. This path exactly matches the path recovered by Boujou. It does not lie entirely on the envelope but partly inside it as it is drawn using the distance of the viewpoint. The path $P(\underline{v}, \underline{d}, \underline{t})$ has a corresponding projection of this trail on the envelope of the view space.

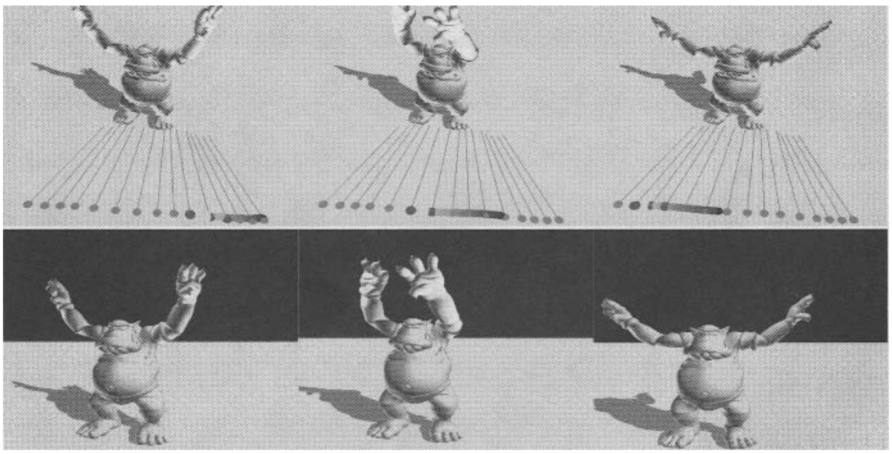


Fig. 4.10. The top row shows the camera path. The bottom row shows the corresponding generated animation frames (see colour insert).

Hence, we can generate view-dependent animations from video. We see, however, that the information content from one video maps to a single path on the view space. We show in the next section that it is possible to generate an animation by tracing other paths on the envelope when multiple view directions with their corresponding poses are defined for a character at a given instant in time. These multiple view directions may be obtained, for example, from multiple synchronized video sources or from augmentation of the view space with sketches.

4.6 Generating the Animation from Multimodal Inputs

We have shown how we can use a video to create a view space and generate a view-dependent animation. We now show how the animator can integrate inputs from sketches and video using the framework to generate a new animation. Suppose the animator wants to replicate the camera movement of a master cinematographer from some existing movie (video) in the new animation. She, however, wants to give a unique movement style to the character and hence wants to pose the character independently, using sketches.

The first step is to recover the camera path from the video and the character poses from the sketches.

4.6.1 Recovering the camera path from video and the character poses from sketches

The video we choose for constructing the example is the famous ballroom dance sequence from Disney's *Beauty and the Beast* [129]. In this sequence the camera moves in a complex downward spiral. We track the camera path, after masking out the moving characters, as explained earlier in Section 4.3.1. Figure 4.11 shows the recovered camera path. Note that we cannot show the actual frame of the video on which camera tracking was done because of copyright issues, but the camera path and feature points shown in Fig. 4.11 are what were actually obtained during our experiments. We want our rendering camera to move along such a camera path in our animation.

The character poses required in the animation are sketched by the animator. We recover the camera corresponding to each sketch and recover the associated character pose, as explained in Sections 3.4 and 3.5 (see Fig. 4.12).

We want to generate the animation by moving along the path recovered from the video. To achieve this objective, we have to transplant the camera path recovered from the video on the view space constructed from the sketches.

4.6.2 Transplanting the camera path on the view space

The cameras recovered from the sketches form a view space. The view space and its envelope is shown in Fig. 4.13(a) with the recovered cameras shown as the small red and blue spheres. The small green sphere denotes current view direction, and the blue spheres are the selected r-closest key views for the current viewpoint. The character's animation is generated by tracing paths on this envelope.

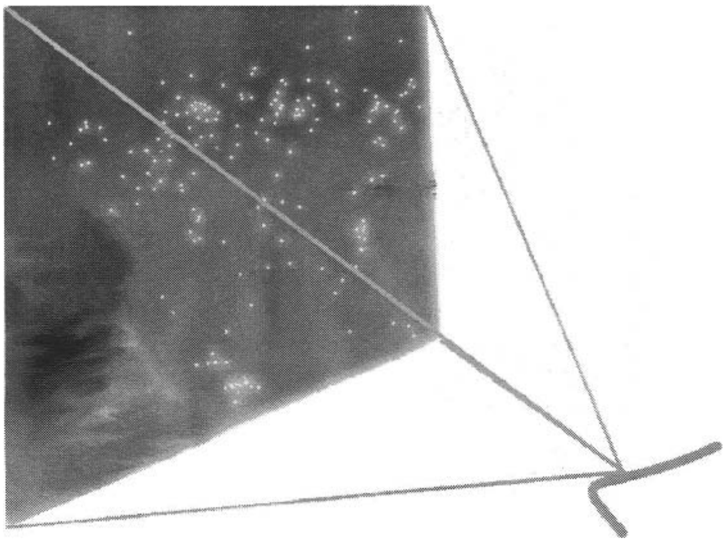


Fig. 4.11. The recovered camera path is seen in the right bottom. The points are the feature points on the background, tracked by Boujou in order to recover the camera path.

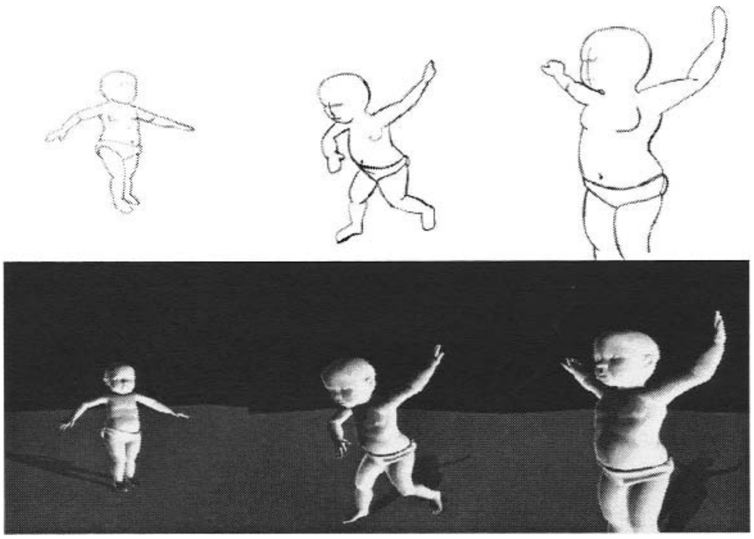


Fig. 4.12. The top row shows some of the sketched poses. The bottom row shows the corresponding poses recovered by the framework.

We want to transplant the camera path recovered from the video sequence onto this view space. The recovered camera path and the view space are in different

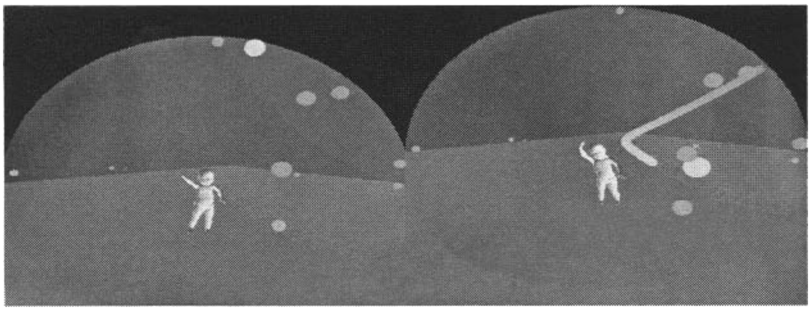


Fig. 4.13. (**a**) envelope of the view space constructed using the cameras recovered from the sketches, (**b**) transplanted camera path (see colour insert).

coordinate systems. The transplantation can be done meaningfully only when we are able to register the two coordinate systems. The two coordinate systems are registered when an average vector from the character to the camera position in the video maps to a similarly oriented character-to-camera vector in view space (see Fig. 4.14). Thus, we want the mapping from one coordinate system to the other to preserve the average relative orientation of the camera with respect to the character. We have a simple algorithm to transplant the camera path into the view space by correctly registering the two coordinate systems with each other (see Algorithm 4.2).

The algorithm thus places the path in the view space in such a manner that the relation of the camera position to the character's position is visually similar to that in the video from which the path is extracted. In this particular example, the character moves on the ground, and since we can track feature points on the ground plane, calculating an estimate of the point A is easy. B is also obtained easily as the average of the recovered camera positions.

Note that the algorithm described above is to assist the animator in placing the transplanted path. The animator can choose to place the path interactively in a different position if it is required, in order to create the desired animation.

As the rendering camera moves along this path, the corresponding animation is automatically generated. The transplanted path [shown as a green trail in Fig. 4.13(b)] in essence matches the path recovered from the video sequence (as shown in Fig. 4.11). The green trail is the locus of all the $q_{\underline{v}}$ at their correct distances along their respective view directions. This demonstrates that the framework is able to handle all the intricate camera variations in terms of view direction and viewpoint distance embodied in the path and generate a new animation.

Thus, we can generate view-dependent animation from multimodal inputs. We can augment a view space by adding more key viewpoints and key poses. Similarly, we want to augment the camera path by extending the transplanted camera path.

```
Require: A known true dimension in the coordinate system in which the
         camera was recovered relative to a dimension in the character
         coordinate system
1  begin
2      Match the scaling of the coordinate system in which the camera path was
       recovered to the scaling of the coordinate system of the character by
       multiplying with a isotropic scaling matrix. This matrix can be
       constructed from the known true dimension.
3      Let $O$ be the origin of the coordinate frame in which the camera was
       tracked.
4      Let $A$ be the average ground position of the character in the video.
5      Let $B$ be the average position of the recovered camera.
6      Let $O'$ be the origin of the coordinate frame in which the view space is
       constructed.
7      Let $A'$ be the average ground position of the 3D mesh model of the
       character.
8      Let $B'$ be the average position of the transplanted camera in the coordinate
       system of the view space.
9      The average character-to-camera vector is given by $\overrightarrow{AB}$.
10     Shift the origin of this coordinate system to $A$.
11     The average character-to-camera vector is now given by $\overrightarrow{OB} \equiv \overrightarrow{AB}$. Shift
       the view space such $O'$ coincides with $A'$.
12     Place the camera path in the view space such that the vector giving the
       average position of the camera, $\overrightarrow{O'B'}$, is the same as $\overrightarrow{OB}$.
13 end
```

Algorithm 4.2: Algorithm to transplant the recovered camera path in the view space.

4.6.3 Augmenting the camera path

The camera path can be augmented by adding new path segments after the transplanted path. The animation sequences that result from the existing and added path segments are seamlessly blended if the added path segment is continuous with the existing path.

We augment the path transplanted in the previous section. The augmented path is shown in Fig. 4.15 along with the generated animation frames.

In this example, the animator has created poses at varying distances along the same view direction. The in-between poses for the animation are generated as explained in Section 4.5 using Algorithm 4.1. An example of this is shown in Fig. 4.16. In the top row the green trail (i.e., q_v) changes only in distance and not in direction with respect to the character. The projection of q_v on the envelope, i.e., $\underline{v}$, represented by the larger green sphere is stationary, indicating that the view direction is constant. However, the corresponding character pose changes as shown in the bottom row are

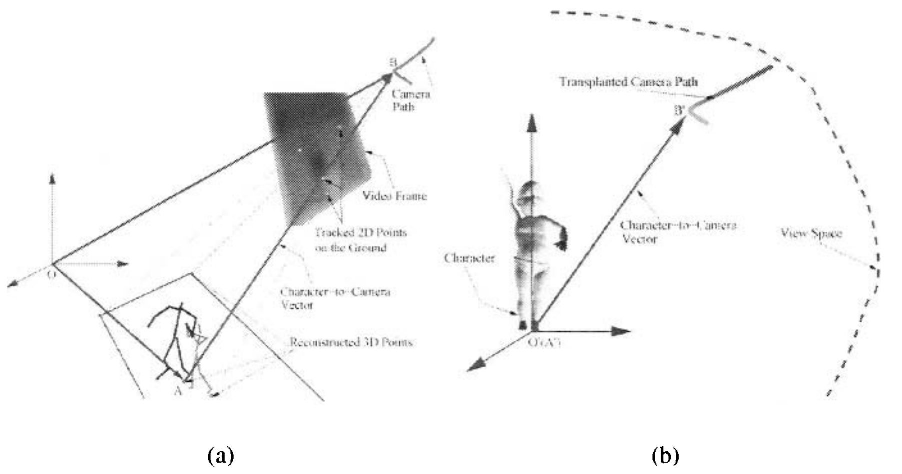


Fig. 4.14. (**a**) Relation between the average (tracked) character position on the ground plane and the average (tracked) camera position, (**b**) shows the same relation between the transplanted path and the posed character.

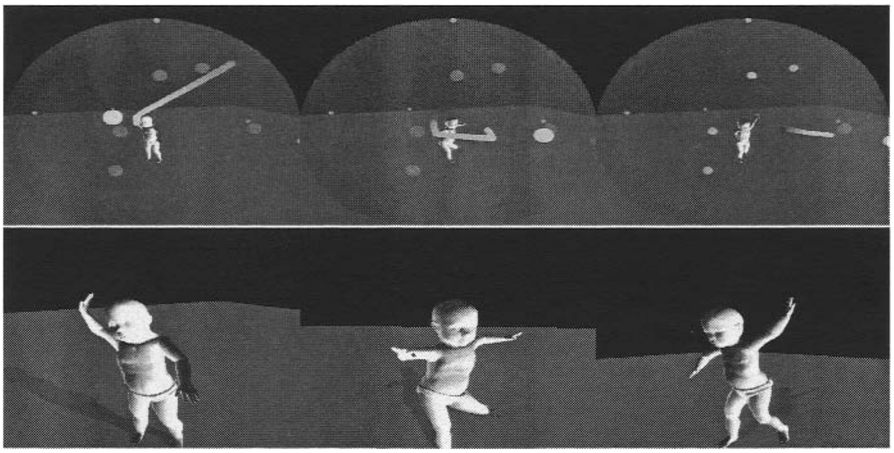


Fig. 4.15. The top row shows the camera path. The bottom row shows the corresponding generated animation frames.

a result of blending caused due to change in distance. The fact that the camera has moved farther away from the character is also apparent in the rendered animation frames shown in the bottom row.

Thus, we have also solved the third problem in multimodal authoring of moving-camera character animations (see Section 4.1). The animator has to trace paths on the envelope of the view space to create the animation. Since all the various input streams

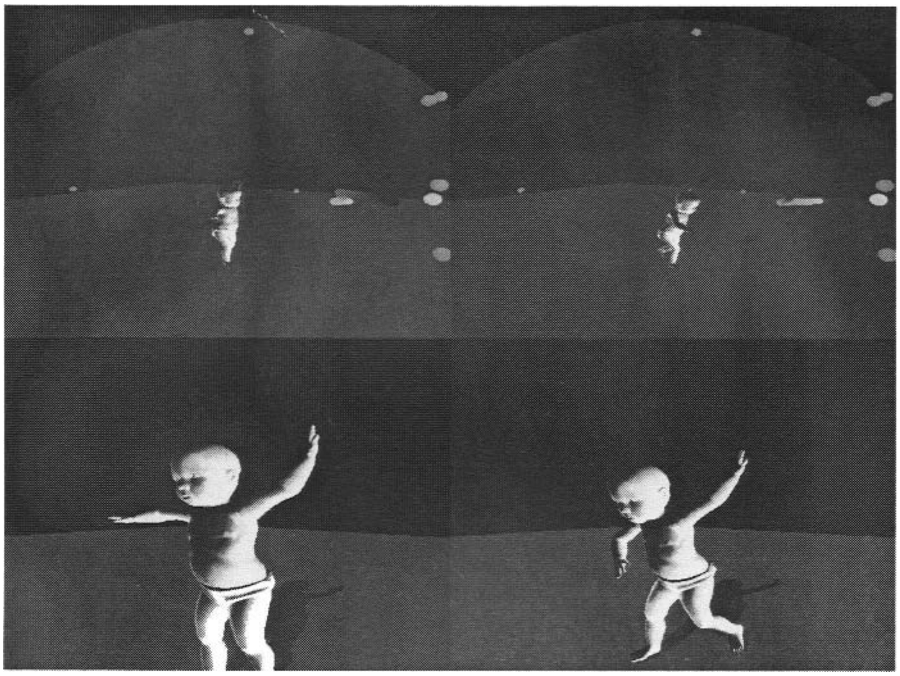


Fig. 4.16. The top row shows the camera path changing only in distance. The bottom row shows the corresponding generated animation frames (see colour insert).

can be represented as the view space, this serves as a common intuitive interface for designing view-dependent animations from these inputs. A path on a complex view space may seem difficult to interpret. All the parameters defining a path, however, are easy to understand because they map to physical camera parameters like camera rotation, translation, and focal length. The animator also gets instant visual feedback, as the animation resulting from the traced out path is displayed in real time.

4.7 Chapter Summary

In this chapter we have shown how to generate view-dependent animation from multimodal inputs. We address the problems inherent to multimodal authoring of view-dependent animations. The primary challenges are to develop techniques to extract relevant information from different types of input, to find a common representation for all this information, and to provide an interface to the animator that can be used to generate the animation. We have presented solutions to each of these problems in this chapter.

We examined the prior work done in the area of video-based animation. In order to do so, we surveyed a variety of methods addressing three stages of a video-based capture system. These are model initialization, tracking, and pose estimation.

Model initialization techniques range from manual initialization to complex pose fitting based on anthropometric data. Tracking methods vary from the use of static thresholding and hidden Markov models to the Kalman filter. Pose estimation techniques use various abstraction models like blobs, contours, or textures. There is limited work done on the capture of the motion of cartoon characters from a video.

We have presented a pipeline to create a view space from video input. The relevant camera and character pose information can be extracted from the video by utilizing camera and character motion tracking techniques. The pipeline has similar higher level structure as the pipeline we developed for sketch-based input in Chapter 3. Pipelines to map other forms of input to a view space can be similarly constructed. Thus, the view space serves as a common representation for all types of inputs.

The method to generate a view-dependent animation from a view space has been demonstrated. We first explained the procedure for generating animations from video inputs. An algorithm is presented that generates the animation, given a camera path on the view space. The animation is generated in real time as the animator traces a path on the view space. This procedure is extended to multimodal inputs.

We have illustrated with an example how to integrate sketch- and video-based inputs for creating a moving-camera character animation. The camera path is extracted from a video, and then the path is transplanted to the view space created from sketches. We have presented an algorithm for automatic transplantation of the camera path. The view space can be augmented with new key viewpoints and poses. We have also demonstrated that it is possible to augment the camera path itself. We extend the transplanted camera path and show that the animations generated from the two segments of the path seamlessly blend together.

5

Stylistic Reuse of View-Dependent Animations

We are now in a position to appreciate the challenges and difficulties faced in creating a moving-camera character animation. We can create view-dependent animations from a variety of inputs using the framework (as seen in Chapters 2 to 4). Here, we consider the different view-dependent animations created by changing the rendering camera, as stylistic variations of each other. We are interested in reusing these variations to synthesize novel animations. We call this process *stylistic reuse*. The view-dependent stylizations can be put to myriad uses in order to synthesize a novel animation.

We present three broad classes of reuse methods. First, we show how to animate multiple characters from the same view space. Next, we show how to animate multiple characters from multiple view spaces. We use this technique to animate a crowd of characters. Finally, we draw inspiration from cubist paintings and create their view-dependent analogues. We use different cameras to control different body parts of the same character and then combine these parts to form a single character in the animation.

Since this chapter of the book bridges across the two themes of stylized animation and animation synthesis, we begin by reviewing the related work pertaining to these two areas.

5.1 Prior Work

Stylized or nonphotorealistic animation and rendering have been used, in recent years, to produce visually appealing imagery. Reuse of stylized animation is difficult because often the stylizations are generated for a particular viewpoint. We present here the prior work done in these two diverse areas.

5.1.1 Stylized animation

Several artwork styles have been explored in stylized animation and rendering literature such as pen and ink, painting, informal sketching, and charcoal drawing. Many of these focus on a specific artistic style or a closely related family of styles. For example, Meier [100] describes a technique for creating animations evocative of impressionistic painting.

Litwinowicz [95] uses image and video analysis to generate impressionistic effects. He allows a user to select size and style of the brush stroke and automatically processes a segment of video. Optical flow and edge detection are used to manipulate a mesh of persistent brush strokes. Additionally, the user can choose to have the brush strokes follow the contours of an image or fix them to a global angle. Hertzmann [56] introduces a method to automatically paint still images with various stroke sizes and shapes, and then extends the technique to video [59]. This work addresses the problem of temporal incoherence in Litwinowicz [95] by selectively updating the properties of brush strokes that lie in video regions that change significantly. However, flickering and scintillation of brush strokes remains a problem. Kowalski et al. [82] create cartoon-style renderings reminiscent of Dr. Seuss.

Hertzmann et al. [58] present a versatile technique to learn nonphotorealistic transformations based on pairs of unpainted and painted example images. These transformations can then be applied to new inputs. This has been extended to video processing by Haro and Essa [51].

In the system described by Kalnins et al. [74], a user draws strokes over 3D models, which are propagated to new frames. DeCarlo and Santella [38] present an algorithm to stylize photographs by running inputs through known image preprocessing algorithms to segment the image into regions that could be given a fixed color. Klein et al. [79] present a tool for generating nonphotorealistic animations from video. This method undertakes a spatio temporal analysis of video. A novel aspect of this work is the use of a set of rendering solids where each rendering solid is a function defined over an interval of time. This allows for the effective and interactive rendering of many nonphotorealistic styles.

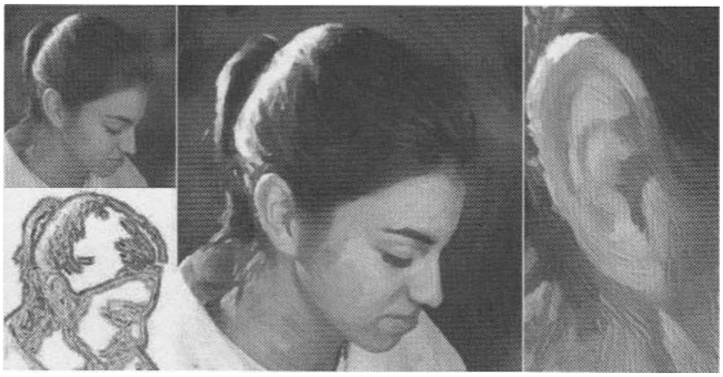


Fig. 5.1. The input image is shown in the left top, the left bottom image shows a visualization of the radial basis function. The output is shown in the center image. A detailed view of the rendered strokes is shown in the right image (image courtesy Hays and Essa [53]).

Hertzmann [57] introduces a new method for adding a physical appearance to brush strokes. The basic idea is to add height fields to brush strokes to allow for lighting calculations. The resulting highlights and shading give the paint strokes a more realistic appearance. Hays and Essa [53] give techniques to transform images and videos into painterly animations depicting different artistic styles. They use radial basis functions to globally interpolate brush stroke orientation (see Fig. 5.1). Temporal incoherency, which was in the past the chief detractor of NPR video techniques, is avoided both by the addition of temporal constraints to previously researched brush-stroke properties and by the addition of new brush stroke properties like opacity. Winnemoller et al. [138] present an automatic, real-time video and image abstraction framework that abstracts imagery by modifying the contrast of visually important features, namely, luminance and color opponency. They reduce the contrast in low-contrast regions using an approximation to anisotropic diffusion and artificially increase contrast in higher contrast regions with difference-of-Gaussian edges.

Glassner [43] talks about using *cubist* principles in animation, i.e., rendering simultaneously from multiple points of view in an animation using an abstract camera model. Stylizations based on innovative use of the camera have also been researched [4, 29].

View-specific distortions also form a very elegant method for specifying stylized animations. View-dependent animation is inherently stylized due to the variations in the animation that occur by changing the parameters of the rendering camera. We want to reuse these stylistic variations to synthesize new animations.

5.1.2 Synthesis and reuse of animation

Recently, there have been a number of projects that allow an animator to create new animations based on motion capture data. Rose et al. [112] use radial basis functions to interpolate between and extrapolate around a set of aligned and labeled example motions (e.g., happy or sad and young or old walk cycles), then use kinematic solvers to smoothly string together these motions.

Brand and Hertzmann [19] describe *style machines* as a technique for stylistic motion synthesis that works by learning motion patterns from a diverse set of motion capture sequences. In the work of Li et al. [92], the data is divided into motion *textons*, each of which can be modeled by a linear dynamic system. Motions are synthesized by considering the likelihood of switching from one texton to the next.

Arikan and Forsyth [7] also present a method for automatic motion generation at interactive rates. Here, the animator sets high-level constraints, like pose and position of a character at specific frames, and a random search algorithm finds appropriate pieces of motion data to concatenate. In a closely related work, the concept of a *motion graph* is defined to control a character's locomotion [81]. The motion graph contains original motion and automatically generated transitions. New motion can be synthesized by building walks on the graph. This allows a user to have high-level control over the motions of the character. It is used for generating different styles of locomotion along arbitrary paths (see Fig. 5.2).

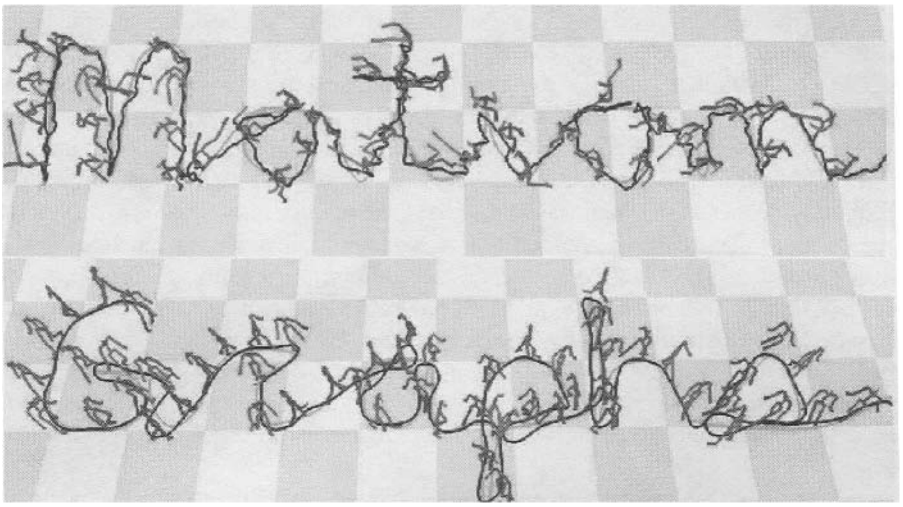


Fig. 5.2. A motion generated to fit to a path that spells "Motion Graphs" in cursive (image courtesy Kovar et al. [81]).

In the work of Lee et al. [88], a new technique is developed for controlling a character in real time using several possible interfaces. The user can choose from a set of possible actions, sketch a path on the screen, or act out the motion in front of a video camera. Animations are created by searching through a motion database using a clustering algorithm. Arikan et al. [8] allow the user to annotate a motion database and then paint a time line with the annotations to synthesize newer motions.

Hsu et al. [63] present a process for transforming an input motion into a new style while preserving its original content. Their system learns to translate by analyzing differences between performances of the same content in input and output styles. It relies on a correspondence algorithm to align motions and a linear time-invariant model to represent stylistic differences. Once the model is estimated with system identification, it is capable of translating streaming input with simple linear operations at each frame.

In recent work, Lee et al. [89] present a technique for allowing animated characters to navigate through a large virtual environment, which is constructed using a set of building blocks. The building blocks are arbitrarily assembled to create novel environments. Each block is annotated with a motion patch, which contains the information about what motions are available for animated characters within the block.

Our technique for synthesizing stylized animation is targeted toward generating stylistic variations of the animations of a character depending on viewpoint changes rather than synthesizing new motion in general. The basis of view-dependent animation provides us a setting to effect such variations very easily. We discuss the representation of such a reuse methodology in terms of the framework (see Chapter 2) in the following sections.

5.2 Animating Multiple Characters from the Same View Space

We want to reuse the view-dependent variations of a character to animate multiple characters and create a novel animation.

Let us assume that a camera, C_1, traces a path, $P_1(\underline{v}, \underline{d}, \underline{t})$, on a view space, $\mathcal{VS}$. A second camera, C_2, traces another distinct path, $P_2(\underline{v}, \underline{d}, \underline{t})$, on $\mathcal{VS}$. The animation generated by C_1 can be thought of as an ordered set of n frames $\mathcal{P}_1$, given by

$$\mathcal{P}_1 = \{\mathrm{p}_1^i : 1 \leq i \leq n\}, \tag{5.1}$$

where p_1^i is the pose of the character in the i-th frame. The order implicitly imposed on the set is the temporal sequence of the frames in the animation. Similarly, the animation generated by C_2 gives another ordered set of m frames $\mathcal{P}_2$,

$$\mathcal{P}_2 = \{\mathrm{p}_2^j : 1 \leq j \leq m\}. \tag{5.2}$$

The animations $\mathcal{P}_1$ and $\mathcal{P}_2$ are view-dependent variations of each other, i.e., they are generated from the same view space. The poses $\mathrm{p}_1^i \in \mathcal{VS}$ and $\mathrm{p}_2^j \in \mathcal{VS}$ are view-dependent variations, or instances, of each other.

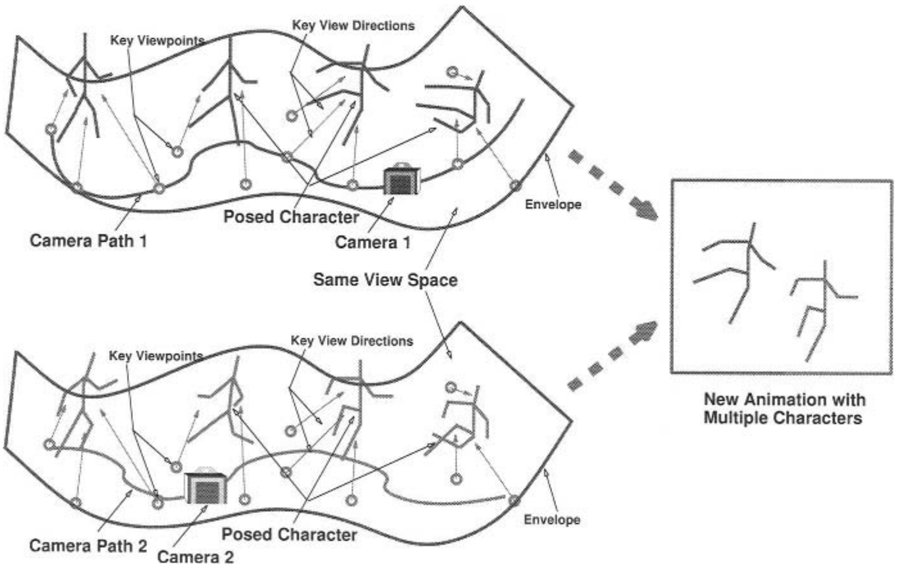


Fig. 5.3. Different camera paths traced on the same view space generate different instances of the animation with different character poses. These different poses can be used to animate multiple characters in a new synthesized animation.

We then define a novel animation with two characters as an ordered set Q, given by

$$Q = \{\langle \mathrm{q}_1^k \oplus \mathrm{q}_2^k \rangle : \mathrm{q}_1^k = \mathrm{p}_1^k \text{ and } \mathrm{q}_2^k = \mathrm{p}_2^k \;\; \forall\, k,\; 1 \leq k \leq \min(n, m)\}, \tag{5.3}$$

where $\langle q_1^k \oplus q_2^k \rangle$ indicates that a frame k in Q consists of two character poses. The $\oplus$ operator indicates that the two poses are being composed together to form the synthesized animation frame. The composition can be done in 3D space if the two poses are registered to a common coordinate system. The composition can also be done in 2D image space by compositing the poses after they have been rendered into the frame buffer (see Fig. 5.3). The novel animation has $min(n,m)$ frames.

Multiple characters may visually obscure each other. If the compositing of the characters is done in the image space, this can be handled by rendering them into different buffers and maintaining separate depth buffer information to resolve occlusion during compositing. If the compositing is done in the object space, then handling occlusion is easier as standard rendering can be used.

In this manner, we can reuse the view-dependent variations of a character to animate multiple characters and create a new animation. As an example of this method of reuse, we create a view space and plan the movement of two cameras on it.

In this particular example, we create a ballet animation for *Hugo*. We create an animation where we want two characters to perform a coordinated ballet sequence; however, both the characters are instances of the same character each performing different ballet moves.

5.2.1 Planning, sketching, and creating the view-dependent models

First the storyboard for the animation is planned. Then the various key poses are sketched, for which the view-dependent models are created later. This stage is guided by the animator's conception of the various poses and camera moves that constitute the animation. A part of the storyboard, illustrating some of planned key frames with the poses of two characters and the rough camera movement, is shown in Fig. 5.4. The red and green arrows show the planned camera movement. The black arrows show the direction in which the character will appear to turn in the video (this movement is because of the camera, which actually moves in the opposite direction). The blue arrows show approximate directions in which the legs of the character will move.

The animator uses the techniques explained in Chapter 3 to recover the cameras from the sketches, which aligns the base mesh with the sketched pose, and to deform the mesh to get the best possible match with the sketches. We show the posed meshes in Fig. 5.5(b) for the sketches shown in Fig. 5.5(a). The poses and the recovered cameras form the view-dependent models, which are then used to create the view space.

5.2.2 Generating animations over a special view space

This example is an instance of the reuse strategy discussed in Section 5.2. However, here we have a view space configuration that makes the implementation of this theory nontrivial. This is because the example requires the character to stand in place and perform certain ballet steps. Thus the view space is an aggregation of view spheres that are separated only in time and not in space (see Section 2.2).

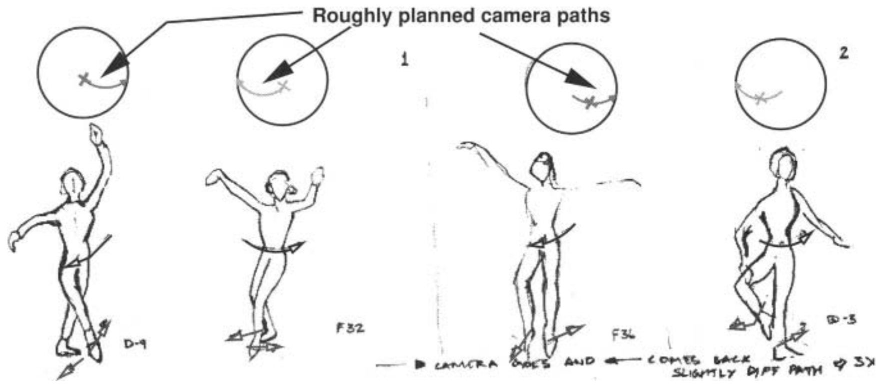

Roughly planned camera paths
1
2
D-9
F32
F36
D-3
CAMERA GOES AND COMES BACK SLIGHTLY DIFF PATH
3X

Fig. 5.4. Part of the storyboard for the reuse animation example (see colour insert).

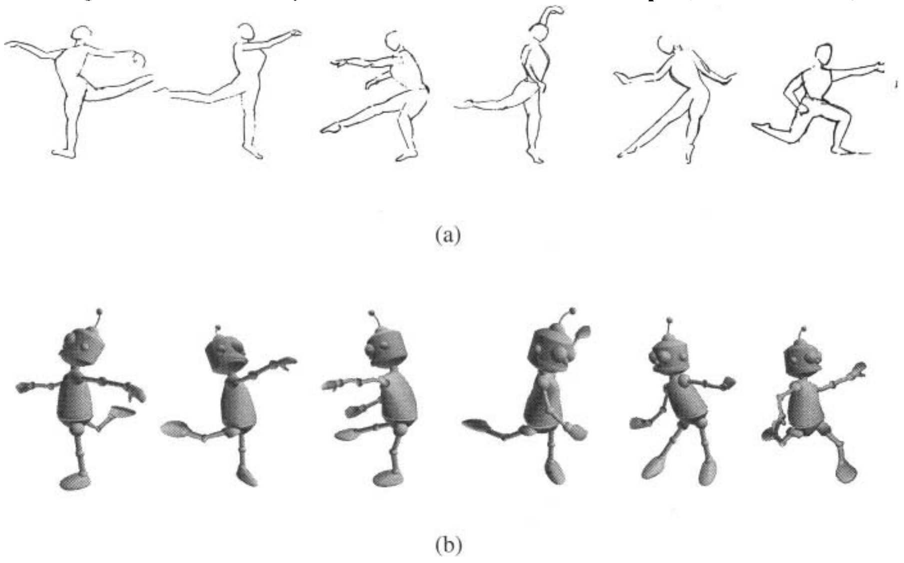


Fig. 5.5. (**a**) some of the sketches for the reuse example, (**b**) corresponding posed meshes seen from the recovered cameras.

We cannot put all the views into a single view sphere because of the following reasons: Such a construction is nonintuitive since it does not capture the temporal characteristics desired by the animator. In this animation, the character has different poses, for the same view direction, at different times. A single view sphere cannot generate such an animation. Also, it causes the planned camera path to be undesirably influenced (during interpolation for getting novel views) by the added views. Multiple view spheres at the same point in space separated in time are perfectly representable within our general framework (see Section 2.2); however, their implementation requires extra effort in order to generate an animation over such a view space.

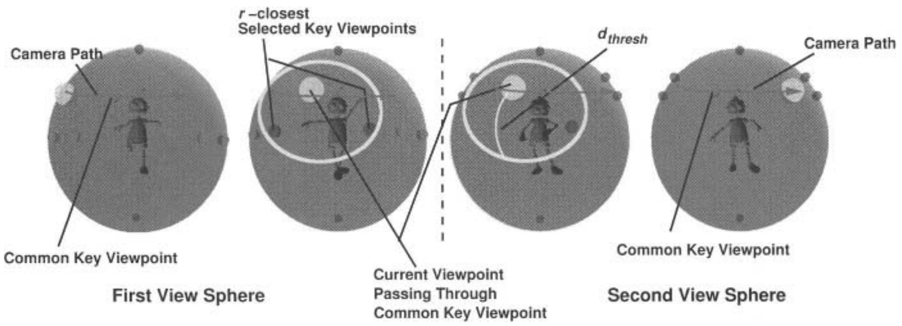


Fig. 5.6. The camera transitioning from one view sphere to another when the point of transition is a key viewpoint.

We move across four view spheres during the animation. The challenge in working with multiple view spheres lies in creating a seamless blend between the resulting animations as the viewpoint is transferred from one view sphere to another. Let VS_1 and VS_2 be the two view spheres, and let us consider the case of transitioning from VS_1 to VS_2. The converse case is symmetric. Let $V = \{v_i : 1 \leq i \leq \mathcal{K}\}$ and $V' = \{v'_j : 1 \leq j \leq \mathcal{K}'\}$ represent the set of key viewpoints of the two view spheres, respectively. Let the point of transition from VS_1 to VS_2 be v_c such that $v_c \in VS_1$ and $v_c \in VS_2$. Note that the camera path has to pass through v_c, by definition. We have the following cases for the point, v_c:

1. The common point, v_c, is a key viewpoint, v, and the key poses associated with v in the two view spheres are the same, $m^1_v \equiv m^2_v \equiv m_v$. In such a case the transition is very easy. When the camera reaches v_c, we simply swap the current view sphere from VS_1 to VS_2.
2. The common point, v_c, is a key viewpoint, v, and the key poses associated with v in the two view spheres are different. We use the weighing function given in Equation (3.14) for calculating the blending weights. If d_{thresh} is the distance threshold decided by the animator and $\underline{v}$ is the current camera position, then we swap m^1_v and m^2_v when $d(\underline{v}, v) = d_{thresh}$. We swap the view spheres themselves only when $\underline{v} = v$ (see Fig. 5.6). It should be noted that only swapping the view spheres when the current viewpoint reaches the point of transition will cause a discontinuity in the animation. We need to swap the key poses as soon as the current camera enters a region where the pose at v begins influencing the current pose.
3. The common point, v_c, is not a key viewpoint. In this case we compute the pose at v_c in VS_1 and VS_2 using the technique explained in Sections 2.2 and 2.3. We then treat v_c as if it were a key viewpoint and repeat the procedure explained in

the first step. Once the camera transitions into the new view sphere, v_c ceases to be a key viewpoint and the animation proceeds as usual.

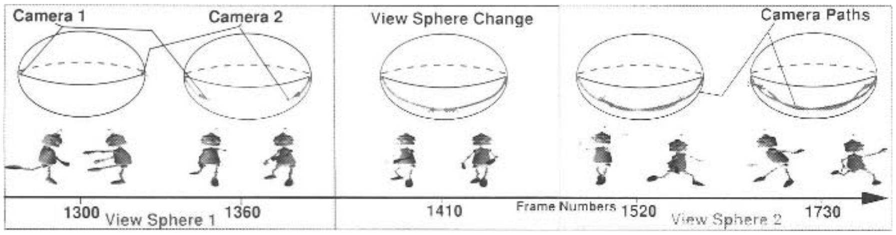


Fig. 5.7. Camera 1 *(in green)* generates the green character, Camera 2 *(in red)* generates the red character. Each view sphere generates two character poses in response to the two cameras (see colour insert).

Hence, once we are able to transition smoothly across the view spheres, we can then generate $\mathcal{P}_1$ and $\mathcal{P}_2$ easily. Then the novel animation can be synthesized according to Equation (5.3). We, however, still need to define the compositing method.

5.2.3 Rendering the animation

To get the desired animation, we plan the movement of two cameras across the view spheres. Each of the cameras will generate one of our characters respectively (see Fig. 5.7). Note that both the cameras cross over from one view sphere to another in the center view sphere while maintaining a smooth camera path.

We composite the poses in 2D image space while transferring them to the framebuffer, to get one coherent dance sequence. We show a sequence of frames from our final animation in Fig. 5.8. The characters in green and red are our two dancers.

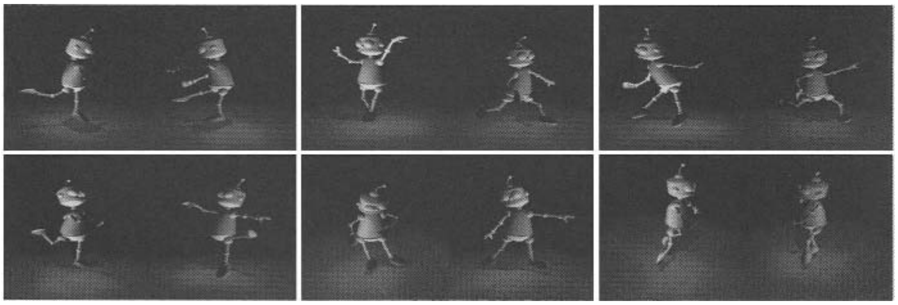


Fig. 5.8. Frames from the synthesized animation (see colour insert).

5.3 Animating Multiple Characters from Multiple View Spaces

The reuse strategy presented in Section 5.2 uses multiple instances of the same character, each from the same view space. We want to further expand this idea and look at animating groups of distinct characters together.

Consider that we have N distinct characters and we have constructed a view space for each. Then we can generate the distinct animations, $\mathcal{P}_r$, with $1 \leq r \leq N$. The generated $\mathcal{P}_r$'s are distinct even if the path traced on each view space is the same because the character in each view space is distinct. Each $\mathcal{P}_r$ is an ordered set of n_r frames and is given by $\mathcal{P}_r = \{\mathrm{p}_r^i : 1 < i \leq n_r\}$. A new animation of a group of these distinct characters can be constructed as

$$Q = \{\langle \bigoplus_{l=1}^{N} \mathrm{q}_l^k \rangle : \mathrm{q}_l^k = \mathrm{p}_l^k \ \forall\, k,\ 1 < k \leq \min_{l=1}^{N}(n_l)\}\,, \tag{5.4}$$

where the $\bigoplus$ operator indicates that N poses are being composed together to form the synthesized animation frame.

We now look at the problem of how to control the paths we want to trace on the N distinct view spaces. Let a camera be associated with every view space. We call this camera the *local* camera for the corresponding view space. Let the path traced by this camera be $P_r(\underline{v}_r, \underline{d}_r, \underline{t}_r)$. We define a single *global* camera and the path traced by this camera as $\mathfrak{P}$. This path $\mathfrak{P}$ is the trajectory of the global camera in 3D space, defined in the global coordinate system. It is not a path on any of the view spaces.

We can define a *path-mapping* function f_r, $P_r = f_r(\mathfrak{P})$, $1 \leq r \leq N$. The function f_r maps the global path to the corresponding local path on the view space. The function f_r is a coordinate system transfer function from the global coordinate system to the local coordinate system of each view space. In order to create the novel animation, the animator has to plan the camera trajectory only for the global camera and define the various f_r's. Then moving the global camera along $\mathfrak{P}$ will cause each of the local cameras to move along the corresponding P_r on their respective view spaces. This will generate the distinct animations, $\mathcal{P}_r$'s. These can be composited together to generate the final animation (see Fig. 5.9). A straightforward choice for the compositing method is to render the various poses as they appear when viewed through the global camera. This technique automatically composites them in the rendered frame. The animator, however, can use any other compositing method as required for the animation. Before starting the animation process, the animator has to place the various characters in the global coordinate system as a part of global scene definition. Hence, the animator already knows the coordinate system transfer function g_r from the global coordinate system to the local coordinate system of each character. The mapping from the local coordinate system of the character to the coordinate system of the view space h_r is easily recovered during view space construction. Thus, we have $f_r = g_r \circ h_r$ (where $\circ$ represents function composition).

We have used this reuse technique to animate a crowd of characters in the *Mexican Wave* animation. In this example, the same character is replicated many times to generate a crowd [shown in Fig. 5.10(b)]. Each character has a local view space as

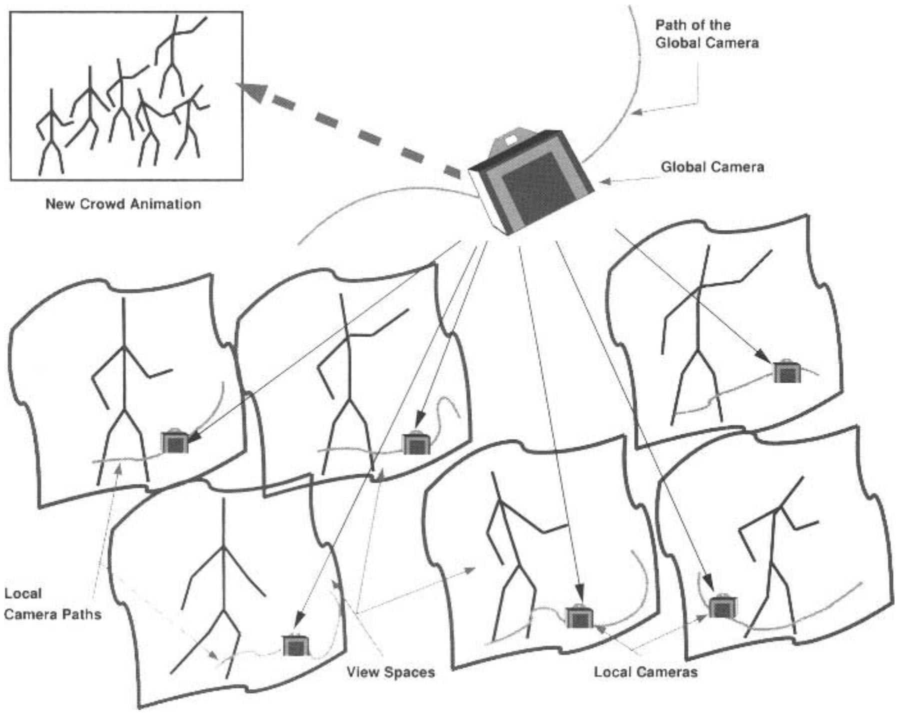


Fig. 5.9. Animating multiple characters from multiple view spaces.

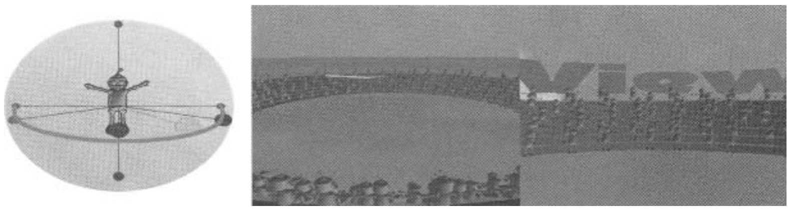


Fig. 5.10. The *Mexican Wave* animation (see colour insert).

shown in Fig. 5.10(a). The local key viewpoints are shown in blue and red, while the current local camera is shown in green. Moving this local camera on the path shown (in green) causes the single character's pose to change as it is supposed to change during the crowd animation. The movement of the global camera is mapped to each of these view spaces to move the corresponding local cameras, which generates the final animation. The path of the global camera and current look-at is shown in green in Fig. 5.10(b). Note that the crest of the Mexican wave is in front of the current camera look-at. The amplitude of the wave decreases as one moves away from the current look-at vector of the global camera in either direction. This is because the mapping function has been designed to curtail the wave at the boundary of the current view frustum. We also perform conservative view culling to efficiently render the crowd. One of the frames of the final animation is shown in Fig. 5.10(c). The animation of a single character due to camera movement in a local view space, the generation of the crowd, and its animation due to the global camera movement are shown in a video.

In this example, it is not difficult to find the path-mapping function, f_r, that will generate the wave in the crowd for a specific movement of the global camera. Figure 5.11 shows the position of the local cameras in their respective local view spaces for a given position of the global camera. The mapping ensures that the local cameras in local view spaces outside the bounds of the current view frustum do not move. This mapping function can be intuitively constructed. For a general case, however, designing a path-mapping function to get a desired animation may not always be easy.

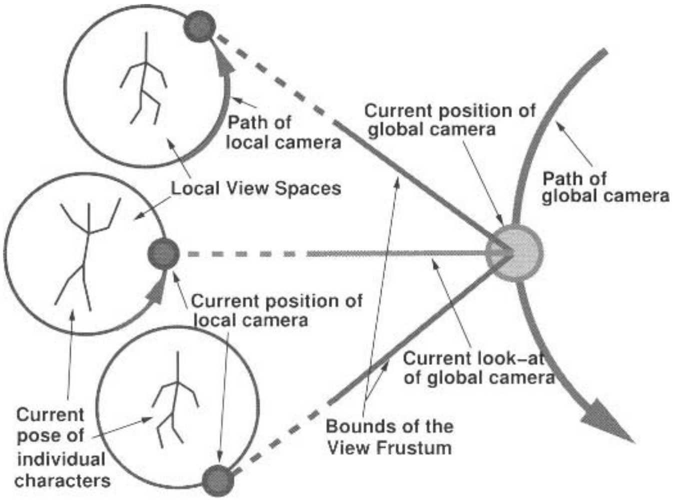


Fig. 5.11. Mapping the movement of global camera to the local cameras.

5.4 Animating Different Parts of a Single Character from a Single View Space

In the previous sections, we looked at the problem of synthesizing a novel animation with multiple characters using view-dependent variations of one or many characters. Now we draw inspiration from *cubist* paintings, which portray the parts of the same character in a painting from different perspectives. Paintings by Pablo Picasso, for example, the *Femme Nue Accroupie*(1959), typify this style of compositing of different views of different parts of a scene into a single projection. They are a perfect example of a scene that can be visually thought of as broken into disjoint parts that are viewed from different perspectives and then patched back together. Similarly, we want to generate a new animation where different parts of the same character are controlled by separate cameras. All the cameras move on the same view space. The final animation will have the character with each separately animated body part blended together.

In order to do this we consider a pose, $\mathfrak{p}$, to be made up of a union of M body parts, $\mathfrak{b}_u$, i.e., $\mathfrak{p} = \bigcup_{u=1}^{M} \mathfrak{b}_u$. We assume there is no overlap between the body parts, i.e., they are distinct and mutually exclusive. Now, we associate a camera C_u with a body part $\mathfrak{b}_u$. Each camera traces a path, $P_u(\underline{v}_u, \underline{d}_u, \underline{t}_u)$, on the view space. The synthesized animation of n frames, Q, is then given by

$$Q = \{\mathfrak{q}^i : \mathfrak{q}^i = \mathfrak{p}^i : 1 \leq i \leq n\} . \tag{5.5}$$

At any point p_u on a camera path, the configuration of the corresponding body part, $\mathfrak{b}_u$, is computed by using a process analogous to pose computation at p_u for a normal view-dependent animation as given in Section 2.2. We can also associate other parameters, e.g., scaling of each body part, with their respective cameras. We can then vary these parameters when their corresponding cameras move. The various body parts are then composited together to form the final pose (see Fig. 5.12). The compositing method used is the animator's prerogative.

We present two variations of this reuse technique as examples. In the first, different body parts are viewed from their respective cameras, and the views are composited in 2D image space to generate a multiperspective image. This compositing technique is similar to the one given by Coleman and Singh [29]. We associate six *body* cameras, one each with the head, torso, two arms, and two legs. We explicitly associate the cameras with the bones of the embedded skeleton for each body part. This automatically groups the mesh vertices into various body parts, as each mesh vertex is uniquely contained in a control lattice cell, which in turn is associated to exactly one bone of the embedded skeleton. We also associate scaling parameters of the various body parts with the position of their respective body cameras. Since each body camera is at a different position, each body part is scaled differently, in addition to having a different perspective. We then composite the view from each to get the final image. In Fig. 5.13(a), the head of the character is seen from the right, the torso from the front, the left hand from the top, the right hand from the left bottom, the left foot from the front, while the right foot is seen from the right side. In Fig. 5.13(b), the head of the character is seen from the left bottom, the torso from the right top, the left hand from the front, the right hand from the top, the left foot from the right side, and the right foot from the left side. This may be thought of as the view-dependent analogue of cubist paintings.

In the second variation, we again associate six body cameras with the various body parts. The composition of the body parts is, however, done in object space, i.e., in 3D. This is done by taking one model of the character and posing the various

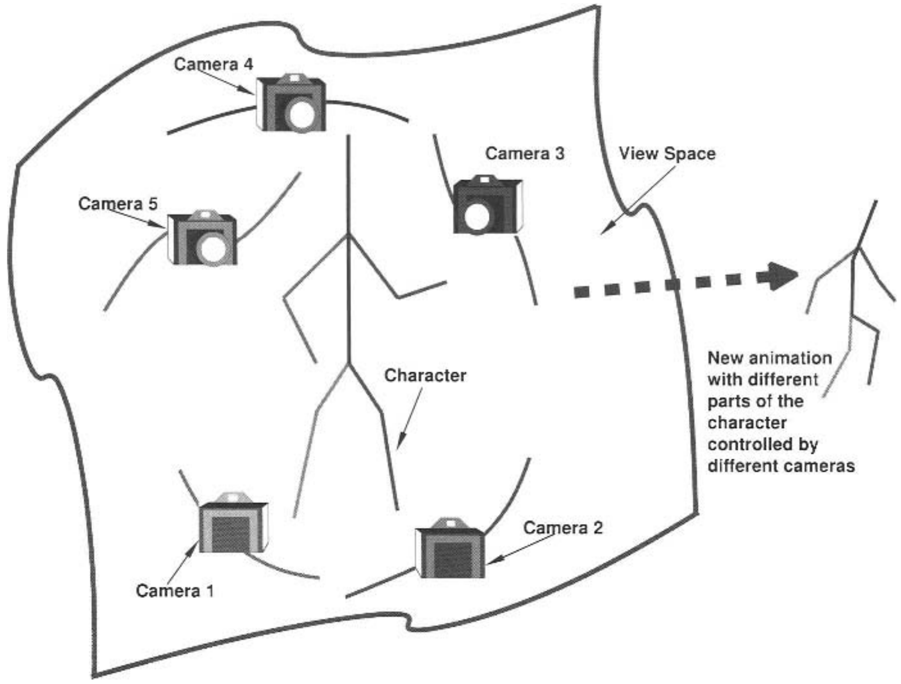

Camera 4
Camera 3
View Space
Camera 5
Character
New animation with different parts of the character controlled by different cameras
Camera 2
Camera 1

Fig. 5.12. Animating different parts of a single character from a single view space.

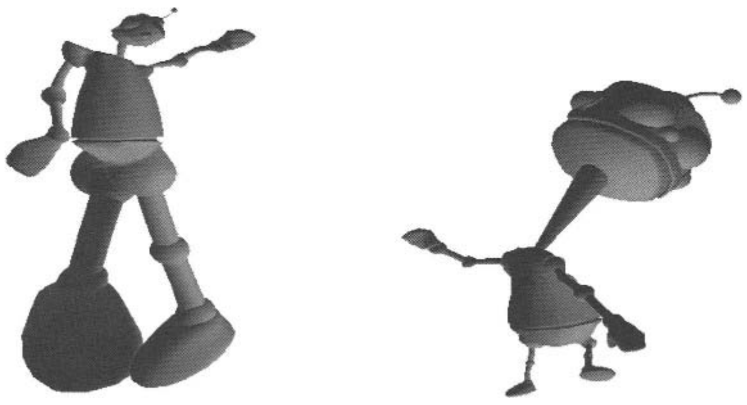


Fig. 5.13. Two examples of multiperspective images.

body parts as per the associated camera. The connectivity of the body parts is not disturbed, and hence they can be blended in object space. The animation is rendered from the viewpoint of a master camera. The body cameras follow the movement of the master camera. Figure 5.14 shows frames from the three animations we have generated using this technique, each with a different set of scaling parameters for the various body parts. In the first case [see Fig. 5.14(a)] there is no scaling applied to each body part; so as the master camera moves, the various body parts are posed as per the key views and the effect is similar to a normal view-dependent animation. In the second case [see Fig. 5.14(b)] scaling applied to each body part is such that it exaggerates the perspective effect, i.e., the part that is closer to the camera appears very big, while the part that is farther away appears very small. This effect can be seen in the legs and the head as the camera moves from below the character to the top. As the camera moves, the scaling associated with the body parts changes to maintain the exaggerated perspective effect. The hands and the torso are not scaled. In the third case [see Fig. 5.14(c)], the scaling applied counters the perspective effect, i.e., body parts that are farther appear larger.

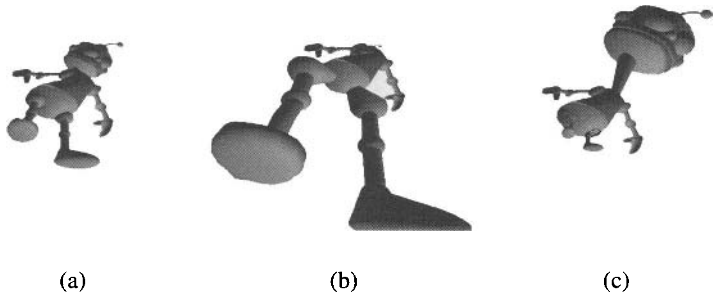


Fig. 5.14. Compositing in object space and rendering from the master camera.

As an example of the elegance of our reuse technique, we stylize the *Hugo's High Jump* animation by associating different cameras with different body parts of the character. Sample frames from this animation are shown in Fig. 5.15. In this animation, as Hugo jumps, his limbs stretch and his head becomes larger. This is made possible by the scaling parameters associated with the various moving body cameras. As Hugo falls down, he resumes his original proportions. In this example also, the body cameras follow the movement of one master camera.

5.5 Chapter Summary

In this chapter, we explored the different possibilities in reusing view-dependent animations to synthesize novel animations. We discussed the state of the art of stylized

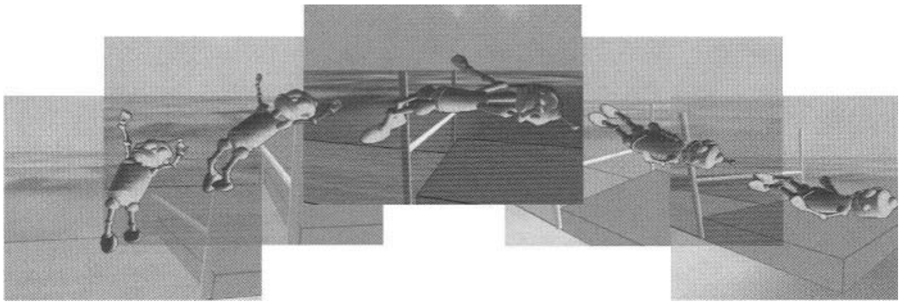


Fig. 5.15. Frames from the stylized *Hugo's High Jump* animation.

animation. We find a large body of literature on nonphotorealistic animation with emphasis on creating particular rendering styles like pen and ink, charcoal drawing, and impressionist paintings. Considerable work has been done in the area of creating nonphotorealistic versions of still images and videos. We also review the work done on animation synthesis from motion databases. Many of these algorithms, like motion graphs, search a database of recorded motion and generate a smooth sequence comprising short animation clips. The graph encodes the transition from one clip to another, and the search is pruned using various constraints and heuristics.

We then presented our techniques for stylistic reuse of view-dependent animations. We introduced three novel reuse strategies. First, we showed how to animate multiple characters from the same view space. Next, we showed how to animate multiple characters from multiple view spaces. We used this technique to animate a crowd of characters. We have drawn inspiration from cubist paintings and created their view-dependent analogues by using different cameras to control various body parts of the same character. Thus, we have shown that reusing view-dependent animation is possible using the framework and it can be used to synthesize a variety of interesting stylized animations. We demonstrated the efficacy of the framework for stylistic reuse by generating complex animations.

We believe that the stylistic reuse of view-dependent animations can lead to the creation of many interesting animations easily and efficiently.

6

Discussion and Future Directions

6.1 Discussion

In this book, we have presented a framework for creating moving-camera character animations. It is often arduous for the animator to manually stage a character's action when the point of view changes in each frame. We have shown that view-dependent animation offers a natural solution to this problem. Since in view-dependent animation the character's action *depends* on the view, the camera and character pose association, once specified by the animator, is maintained throughout the animation. In the course of designing a general framework that encapsulates the rich diversity offered by moving-camera animations, we have solved many challenging problems.

We present a concise summary of the features of the framework for view-dependent character animation.

1. We have formulated the concept of a view space of key views and associated key character poses. This provides a formal theoretical basis for representing view-dependent animations and forms the core of the framework.
   - The view space representation captures all the information about the views and character poses efficiently and concisely.
   - The animator can trace camera paths on the view space, and the corresponding animation is generated in real time. Simple interpolation schemes are used to generate in-between character poses from the space of key poses while the rendering camera moves on the path specified by the animator.
2. The view space embodies all the information contained in the various view (or camera) parameters. Robust computer vision techniques have been used to estimate these parameters from different types of inputs.
   - The algorithms used to recover cameras from sketches are numerically robust and efficient. We are able to recover a wide variety of cameras, ranging from the orthographic to the full projective, from a sketch. This allows us to reproduce the viewpoint intended in the sketch more faithfully in the animation. The algorithms are resilient to many types of sketches, which can be

  mannequin sketches, stick-figure sketches, or more accurate sketches of the character.
- The algorithm used to track the cameras in video input, is stable. The cameras recovered by the tracker generate a view space. We show that for a given video we get a single camera path on this view space, which reproduces the camera movement in the video.

3. We have presented a pipeline to extract cameras and character poses and generate a view space from sketches. In order to pose the character from a sketch, we have developed two novel view-dependent algorithms. These allow us to embed a multilayered deformation system into a view-dependent setting and integrate it with computer vision techniques.
   - The *view-dependent posing algorithm* poses the skeleton embedded inside the mesh model of the character in such a manner that the pose matches the sketched pose when viewed through the recovered camera. The algorithm works at interactive rates and provides instant feedback to the animator. The animator also has the option of using IK directly to manually fine tune the pose recovered by the algorithm.
   - The *view-dependent deformation algorithm* deforms the mesh model of the 3D character such that the silhouette of the mesh model matches the outline of the sketched character when viewed through the recovered camera. The algorithm uses DFFD as a backend to displace the mesh vertices. The solution is pruned space using projection constraints derived from the recovered camera.
   - We chose to use IK and DFFD and have modified and integrated them into the framework. They are sufficient to demonstrate the viability of the framework, while being reasonably efficient at the same time. However, we would like to point out that other alternatives to these techniques exist in recent literature (see Sections. 3.5.1 and 3.5.4). These can be suitably adapted to replace IK and DFFD without affecting the view-dependent posing and deformation algorithms. The implementation of our pipeline in this book should be considered as a concept demonstration.
4. Multimodal authoring of view-dependent animations is a challenging problem. We have presented a solution to this problem and illustrated it using interesting examples.
   - We develop a pipeline to create a view space from multimodal inputs. We demonstrate this process for video input and present arguments to show that similar pipelines can be constructed for other input types.
   - The view space serves as the common ground for all types of inputs and allows the animator to mix the information contained in them to create the desired animation. We have presented an example of generating a view-dependent animation using a video sequence as input.
   - We present an example that demonstrates how it is possible to combine sketch- and video-based inputs for creating a moving-camera character animation. The camera path is extracted from a video and transplanted onto a

view space created from sketches. We have developed an algorithm for automatic transplantation of the camera path by registering the corresponding coordinate systems with each other. We can augment the view space with new key viewpoints and poses. It is also possible to augment the camera path itself by adding new path segments and generate a seamlessly blended animation from the various path segments.

5. The framework is not meant to replace the complete animation pipeline, but rather complement it. View-dependent animation can be seamlessly blended with non-view-dependent animation, and we demonstrate this in many examples.
6. Although we have developed algorithms for automating most of the stages of our work as far as possible, there is still a manual fall-back option for each stage to give the animator adequate control over the animation. The animator can decide to manually tune the output at any stage if so desired.
7. The ability to understand and explore view-dependent animation using the framework gives us an insight into the various applications of view-dependent animation. We formalize the concept of stylistic reuse of view-dependent animations in terms of our framework.
   - We define a synthesized animation as a combination of the animations generated by the view-dependent instances of a single character. The different instances appear as multiple characters in the final animation. An example of this formulation generates a complex ballet animation, which has two view-dependent instances of the same character performing different, yet synchronized, ballet moves.
   - We also formulate two other possible reuse strategies. In the first, we propose that it is possible to generalize our technique to encompass view-dependent animations of a group of different characters. For the second, we have drawn inspiration from cubist paintings and created their view-dependent analogues by using different cameras to control various body parts of the same character.
8. We have implemented the following components of this framework: the inverse kinematics and direct free-form deformation engines, exponential map parameterization of rotations, joint reach cones, Kalman-filter-based contour tracker, camera recovery, and character posing algorithms. We would like to integrate the view-dependent character animation work flow into a commercial animation production pipeline, and have professional animation artists evaluate it for further improvements.

We have shown that view-dependent animation is an easy, efficient, and intuitive solution for creating moving-camera character animation. We, however, feel that there is a lot of potential in the method, which can be harnessed to solve a variety of other problems. In the next section, we discuss some interesting problems for future work.

6.2 Future Directions

We discuss some interesting problems which can be solved using our framework and methods for extending or improving it.

Integrating motion capture and multiple video streams: We have already argued theoretically that the framework can represent information contained in many different forms of input using the view space. It would be interesting to practically implement these alternate pipelines, to experiment with the myriad, interesting animations that can then be generated.

Optical motion capture setups have camera information because all the cameras are calibrated. The poses of the characters are recovered from the motion capture data. Since the character pose and camera association is inherent to the motion capture setup, it naturally maps to a view space. Similarly, multiple synchronized video streams can be used to recover 3D pose information of characters. Each of these videos can be mapped onto a different path on a view space.

Once a view space has been created, the framework offers tremendous creative freedom to the animator, as having one coherent representation for all the information from these myriad sources allows the animator to mix and match them very easily.

View-dependent animation of multiple characters: We have demonstrated our techniques using many examples. All these, however, feature a single animated character. The view space formalism is not restricted to a solitary character. Multiple characters can be associated with a key view in a view space. Tracing a camera path will then animate all these characters together. Another way to extend this formalism is to associate the configuration of the complete scene (i.e., all objects in the scene) with the key views (i.e., adopt a *scene-centric* approach rather than a *character-centric* one). Then, moving the camera would result in an animation with one scene configuration changing into another.

View-dependent timing of animation: In this work, we have not explicitly dealt with the issue of animation timing. The timing in our animations is derived from the sampling of the camera positions on the camera path. Timing, however, is one of the very fundamental principles of animation. It is often seen that timing in close-up shots is different from the timing in medium- or long-range shots. It can be argued that in a close-up shot, an animator wants to show some detail, and hence, the close-up view is timed slower. A sweeping shot of a landscape showing the character running at a high speed from faraway will have timing different from a zoom-in shot that follows the character closely during the run. Hence, we can associate the timing of an animation with the camera in such cases. This is the motivation behind exploring a view-dependent timing strategy. If every point of the view space can be painted with a timing attribute, then the rendering camera will automatically have timing information associated with it. The challenge, however, is to develop an intuitive interface for the animator to specify this timing information on the view space. It will also be necessary to resolve timing inconsistencies that may arise due to arbitrary camera paths on the view space.

View-dependent lighting and texture: We dealt with 3D character animation in this entire work. Current animation productions, however, use a hybrid of 2D and 3D techniques for character animation. Cooper describes [31] how animators at Dream-Works used a traditionally drawn 2D animated horse for the primary character in the movie *Spirit: Stallion of the Cimarron* [9] and merged it seamlessly into beautiful 3D sets and camera moves. The lighting and tonal texture, in both 2D and 3D, are dependent on the final rendering camera. Hence, they have a "view-dependence" property, which can be modeled using the framework.

The above mentioned directions of future work are some of the ways in which the framework can be used and extended. View dependence as a property, however, has been exploited in various areas. Some of these include creating and rendering multiresolution and progressive meshes [62], generating levels of detail (LOD) for complex scenes [94], fast displacement mapping [133], and point-based nonphotorealistic rendering [33]. All of these can be associated with the view space, which essentially embodies the concept of view dependence. These open many interesting and different directions for future work.

A

Camera Models and Computation of the Camera Matrix

A camera is a mapping between the 3D world (object space) and a 2D image. Here we present in brief a few camera models which are matrices with particular properties that represent the camera mapping. We also present two algorithms to compute the projective and affine cameras given a set of point correspondences.

A.1 The Pinhole Camera Model

Consider a central projection of points in space onto a plane. Let the center of projection be the origin of a Euclidean coordinate system, and consider the plane $z = f$, which is called the *image plane* or *focal plane* (see Fig. A.1). Under this pinhole camera model, a point $\mathbf{X} \equiv (X, Y, Z)^{\mathsf{T}}$ is mapped to a point $\mathbf{x} \equiv (fX/Z, fY/Z, f)^{\mathsf{T}}$ on the image plane. Then if the world and image points are represented by homogeneous vectors, the central projection is expressed as a linear mapping given by

$$\begin{pmatrix} fX \\ fY \\ Z \end{pmatrix} = \begin{bmatrix} f & & & 0 \\ & f & & 0 \\ & & 1 & 0 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} . \tag{A.1}$$

Equation (A.1) assumed that the origin of the camera coordinate system is at the origin of the world coordinate system. It also assumed that the world Z axis is the principal axis of the camera. In general, points in space are expressed in terms of a different Euclidean coordinate frame than the camera coordinate system. These world and the camera coordinate systems are related via a rotation, $\mathbf{R}$, and a translation, $\mathbf{t}$. In Equation (A.1) we also assumed the image plane origin coincides with the principal point. This need not be the case always. In the most general case, a 3×4 *camera projection matrix*, $\mathbf{P}$, can be decomposed as

$$\mathbf{P} = \mathbf{K}[\mathbf{R}|\mathbf{t}] . \tag{A.2}$$

Here the matrix $\mathbf{K}$ maps the points from the camera coordinate system to the image coordinate system.

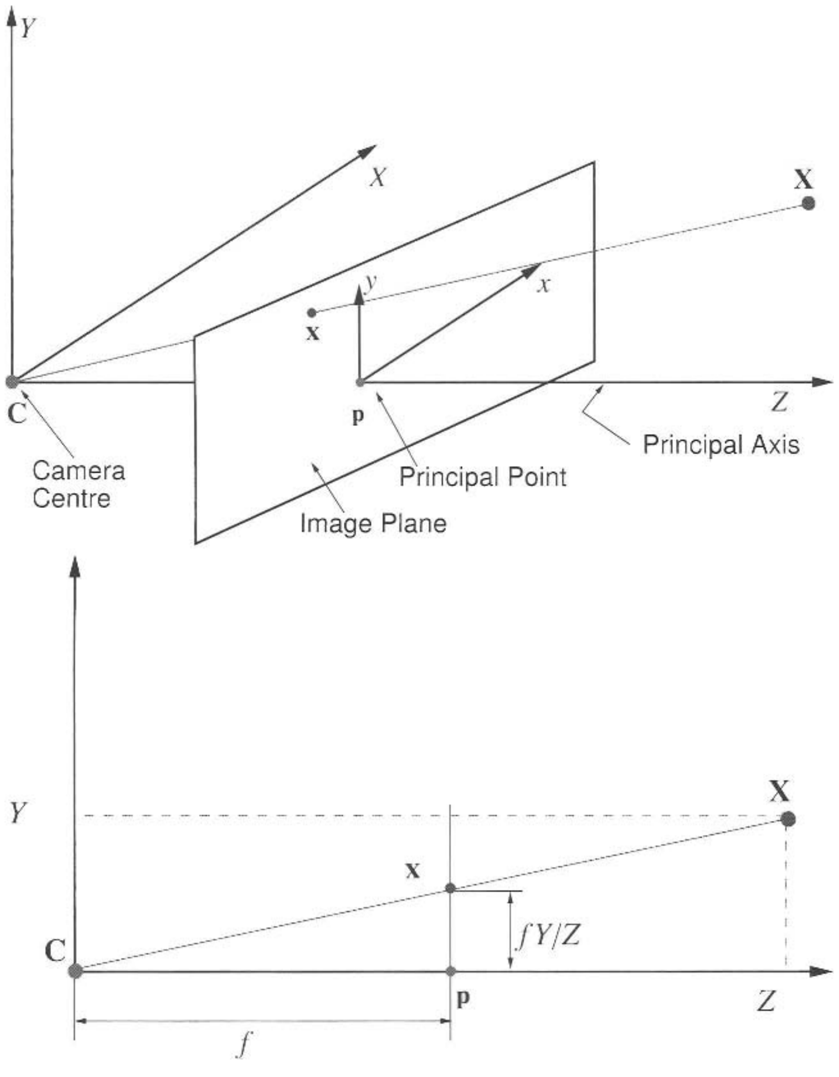


Fig. A.1. Pinhole camera geometry: **C** is the camera center and **p** the principal point. Here the camera center is placed at the coordinate origin.

A.2 Anatomy of the Projective Camera

A general projective camera may be decomposed into blocks according to $\mathbf{P} = [\mathbf{M}|\mathbf{p}_4]$, where $\mathbf{M}$ is a 3×3 matrix. We now have the following properties for a projective camera:

Camera center: The camera center is the right null space $\mathbf{C}$ of $\mathbf{P}$, i.e., $\mathbf{PC} = 0$. For finite cameras ($\mathbf{M}$ is not singular) we get

$$\mathbf{C} = \begin{pmatrix} -\mathbf{M}^{-1}\mathbf{p}_4 \\ 1 \end{pmatrix} . \tag{A.3}$$

Column points: For $i = 1, 2, 3$, the column vectors $\mathbf{p}_i$ are vanishing points in the image, corresponding to the X, Y, and Z axes, respectively. Column $\mathbf{p}_4$ is the image of the coordinate origin.

Principal plane: The principal plane of the camera is $\mathbf{P}^3$, the last row of $\mathbf{P}$.

Axis planes: The planes $\mathbf{P}^1$ and $\mathbf{P}^2$ (the first and the second rows of $\mathbf{P}$) represent planes in space through the camera center, corresponding to points that map to the image lines $x = 0$ and $y = 0$, respectively.

Principal point: The image point $\mathbf{x}_0 = \mathbf{M}\mathbf{m}^3$ is the principal point of the camera where $\mathbf{m}^{3^\top}$ is the third row of $\mathbf{M}$.

Principal ray: The principal ray (axis) of the camera is the ray passing through the camera center $\mathbf{C}$ with the direction $\mathbf{m}^{3^\top}$. The principal axis vector $v = det(\mathbf{M})\mathbf{m}^3$ is directed toward the front of the camera, where $det(\mathbf{M})$ is the determinant of $\mathbf{M}$.

A.3 Cameras at Infinity

An *affine camera* is one that has a camera matrix $\mathbf{P}$ in which the last row $\mathbf{P}^{3\top}$ is of the form $(0, 0, 0, 1)$. For such cameras, $\mathbf{M}$ is singular. The camera center $\mathbf{C}$ is given by

$$\mathbf{C} = \begin{pmatrix} \mathbf{d} \\ 0 \end{pmatrix} , \tag{A.4}$$

where $\mathbf{d}$ is the null 3-vector of $\mathbf{M}$, i.e., $\mathbf{Md} = 0$. The vector $\mathbf{d}$ also gives the direction of parallel projection.

There exists a hierarchy of camera models representing progressively more general cases of parallel projection. These are

Orthographic projection: An orthographic camera has 5 degrees of freedom, namely, three parameters describing the rotation matrix $\mathbf{R}$, plus the two offset parameters t_1 and t_2. The first two rows of the matrix are orthogonal and of unit norm, and $t_3 = 1$.

$$\mathbf{P} = \begin{bmatrix} \mathbf{r}^{1\top} & t_1 \\ \mathbf{r}^{2\top} & t_2 \\ \mathbf{0}^\top & 1 \end{bmatrix} . \tag{A.5}$$

Scaled orthographic projection: A scaled orthographic projection is an orthographic projection followed by isotropic scaling, and is given by

$$\mathbf{P} = \begin{bmatrix} k & & \\ & k & \\ & & 1 \end{bmatrix} \begin{bmatrix} \mathbf{r}^{1\top} & t_1 \\ \mathbf{r}^{2\top} & t_2 \\ \mathbf{0}^\top & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{r}^{1\top} & t_1 \\ \mathbf{r}^{2\top} & t_2 \\ \mathbf{0}^\top & 1/k \end{bmatrix} . \tag{A.6}$$

It has 6 degrees of freedom. The first two rows are orthogonal and of equal norm.

Weak perspective projection: Here the scale factors in the two axial directions are not equal. Such a camera matrix is of the form

$$\mathbf{P} = \begin{bmatrix} \alpha_x & & \\ & \alpha_y & \\ & & 1 \end{bmatrix} \begin{bmatrix} \mathbf{r}^{1\top} & t_1 \\ \mathbf{r}^{2\top} & t_2 \\ \mathbf{0}^\top & 1 \end{bmatrix} . \tag{A.7}$$

It has 7 degrees of freedom, and the first two rows of the matrix are orthogonal.

Affine camera: The general affine camera has an additional skew term, and is of the form

$$\mathbf{P_A} = \begin{bmatrix} \alpha_x & s & \\ & \alpha_y & \\ & & 1 \end{bmatrix} \begin{bmatrix} \mathbf{r}^{1\top} & t_1 \\ \mathbf{r}^{2\top} & t_2 \\ \mathbf{0}^\top & 1 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & t_1 \\ m_{21} & m_{22} & m_{23} & t_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \tag{A.8}$$

A.4 Computation of the Projective Camera Matrix

Given a number of point correspondences $\mathbf{X}_i \leftrightarrow \mathbf{x}_i$ between 3D points $\mathbf{X}_i$ and 2D image points $\mathbf{x}_i$, we want to find a 3×4 camera matrix $\mathbf{P}$ such that $\mathbf{x}_i = \mathbf{P}\mathbf{X}_i$ for all i. If the j-th row of the matrix $\mathbf{P}$ is denoted by $\mathbf{P}^{j\top}$, then we can write

$$\mathbf{P}\mathbf{X}_i = \begin{pmatrix} \mathbf{P}^{1\top}\mathbf{x}_i \\ \mathbf{P}^{2\top}\mathbf{x}_i \\ \mathbf{P}^{3\top}\mathbf{x}_i \end{pmatrix} . \tag{A.9}$$

Writing $\mathbf{x}_i = (x_i, y_i, w_i)^\top$, the cross product $\mathbf{x}_i \times \mathbf{P}\mathbf{X}_i$ can be written explicitly as

$$\mathbf{x}_i \times \mathbf{P}\mathbf{X}_i = \begin{pmatrix} y_i\mathbf{P}^{3\top}\mathbf{X}_i - w_i\mathbf{P}^{2\top}\mathbf{X}_i \\ w_i\mathbf{P}^{1\top}\mathbf{X}_i - x_i\mathbf{P}^{3\top}\mathbf{X}_i \\ x_i\mathbf{P}^{2\top}\mathbf{X}_i - y_i\mathbf{P}^{1\top}\mathbf{X}_i \end{pmatrix} . \tag{A.10}$$

Since $\mathbf{P}^{j\top}\mathbf{X}_i = \mathbf{X}_i^\top\mathbf{P}^j$ for $j = 1 \ldots 3$ and $\mathbf{x}_i \times \mathbf{P}\mathbf{X}_i = 0$, we get

$$\begin{bmatrix} \mathbf{0}^\top & -w_i\mathbf{X}_i^\top & y_i\mathbf{X}_i^\top \\ w_i\mathbf{X}_i^\top & \mathbf{0}^\top & -x_i\mathbf{X}_i^\top \\ -y_i\mathbf{X}_i^\top & x_i\mathbf{X}_i^\top & \mathbf{0}^\top \end{bmatrix} \begin{pmatrix} \mathbf{P}^1 \\ \mathbf{P}^2 \\ \mathbf{P}^3 \end{pmatrix} = 0 . \tag{A.11}$$

We may choose to use only the first two equations because the three equations of Equation (A.11) are linearly dependent. From a set of n point correspondences, we obtain a $2n \times 12$ matrix $\mathbf{A}$ by stacking up the equations for each correspondence. The projection matrix $\mathbf{P}$ is computed by solving the set of equations $\mathbf{Ap} = 0$, where $\mathbf{p}$ is the vector containing the entries of the matrix $\mathbf{P}$, i.e.,

$$\mathbf{P} = \begin{bmatrix} p_1 & p_2 & p_3 \\ p_4 & p_5 & p_6 \\ p_7 & p_8 & p_9 \end{bmatrix} . \tag{A.12}$$

We use an algorithm called the Normalized Direct Linear Transformation(DLT) Algorithm (reproduced from Hartley and Zisserman [52] in Algorithm A.1) to obtain the solution. Algorithm A.1 minimizes $||\mathbf{Ap}||$ subject to $||\mathbf{p}|| = 1$. The residual $\mathbf{Ap}$ is known as the *algebraic error*. Normalizing the data points before performing DLT increases the numerical accuracy of the results and also makes the algorithm invariant to arbitrary choices of scale and coordinate origin. The normalization is better suited to cases where the variation in the depth of points from the camera is relatively less. Since all our 3D data points lie on the character, their distribution in space is compact, and so normalization is the right thing to do. Hartley and Zisserman [52] discuss the numerical stability and error analysis of the DLT in greater detail.

```
Require: Given $n \geq 6$ world to image point correspondences $\{\mathbf{X}_i \leftrightarrow \mathbf{x}_i\}$.
1 begin
2     Normalization of X: Compute a similarity transformation $\mathbf{S}$, consisting of
      a translation and scaling, that takes points $\mathbf{X}_i$ to a new set of points $\tilde{\mathbf{X}}_i$ such
      that the centroid of the points $\tilde{\mathbf{X}}_i$ is the world coordinate origin $(0,0,0)^\top$
      and their average distance from the origin is $\sqrt{3}$.
3     Normalization of x: Similarly compute a similarity transformation $\mathbf{T}$,
      consisting of a translation and scaling, that takes points $\mathbf{x}_i$ to a new set of
      points $\tilde{\mathbf{x}}_i$ such that the centroid of the points $\tilde{\mathbf{x}}_i$ is the image coordinate
      origin $(0,0)^\top$ and their average distance from the origin is $\sqrt{2}$.
4     DLT: Form the $2n \times 12$ matrix $\mathbf{A}$ by stacking up the equations [see
      Equation (A.11)] for each correspondence $\{\mathbf{X}_i \leftrightarrow \mathbf{x}_i\}$. Write $\mathbf{p}$ for the
      vector containing the entries of the matrix $\tilde{\mathbf{P}}$. A solution of $\mathbf{Ap} = 0$, subject
      to $||\mathbf{p}|| = 1$, is obtained from the unit singular vector of $\mathbf{A}$ corresponding to
      the smallest singular value. Specifically this is the SVD of $\mathbf{A}$ gives
      $\mathbf{A} = \mathbf{UDV}^\top$ with $\mathbf{D}$ diagonal with positive entries arranged in the
      descending order down the diagonal; then $\mathbf{p}$ is the last column of $\mathbf{V}$.
5     Denormalization: The camera matrix for the original, unnormalized
      coordinates is obtained from $\tilde{\mathbf{P}}$ as

          $\mathbf{P} = \mathbf{T}^{-1}\tilde{\mathbf{P}}\mathbf{S}\,.$          (A.13)

6 end
```

Algorithm A.1: Normalized Direct Linear Transformation Algorithm.

A.5 Computation of the Affine Camera Matrix

The DLT estimation of the camera in this case minimizes $||\mathbf{Ap}||$ subject to the condition that the last row of the projection matrix $\mathbf{P}^{3\top} = (0,0,0,1)$.

```
Require: Given $n \geq 4$ world to image point correspondences $\{\mathbf{X}_i \leftrightarrow \mathbf{x}_i\}$.
1 begin
2     Normalization of X: Compute a similarity transformation $\mathbf{S}$, consisting of
      a translation and scaling, that takes points $\mathbf{X}_i$ to a new set of points $\tilde{\mathbf{X}}_i$ such
      that the centroid of the points $\tilde{\mathbf{X}}_i$ is the world coordinate origin $(0,0,0)^\top$
      and their average distance from the origin is $\sqrt{3}$.
3     Normalization of x: Similarly compute a similarity transformation $\mathbf{T}$,
      consisting of a translation and scaling, that takes points $\mathbf{x}_i$ to a new set of
      points $\tilde{\mathbf{x}}_i$ such that the centroid of the points $\tilde{\mathbf{x}}_i$ is the image coordinate
      origin $(0,0)^\top$ and their average distance from the origin is $\sqrt{2}$.
4     Each correspondence $\{\mathbf{X}_i \leftrightarrow \mathbf{x}_i\}$ contributes equations
          $\begin{bmatrix} \tilde{\mathbf{X}}_i^\top & \mathbf{0}^\top \\ \mathbf{0}^\top & \tilde{\mathbf{X}}_i^\top \end{bmatrix} \begin{pmatrix} \mathbf{P}^1 \\ \mathbf{P}^2 \end{pmatrix} = \begin{pmatrix} \tilde{x}_i \\ \tilde{y}_i \end{pmatrix}$,        (A.14)
      which are stacked into a $2n \times 8$ matrix equation $\mathbf{A}_8\mathbf{p}_8 = \mathbf{b}$, where $\mathbf{p}_8$ is the
      8-vector containing the first two rows of $\tilde{\mathbf{P}}_A$.
5     The solution is obtained by the pseudoinverse of $\mathbf{A}_8$
          $\mathbf{p}_8 = \mathbf{A}_8^+\mathbf{b}$        (A.15)
      and $\tilde{\mathbf{P}}^{3\top} = (0,0,0,1)$.
6     Denormalization: The camera matrix for the original, unnormalized
      coordinates is obtained from $\tilde{\mathbf{P}}_A$ as
          $\mathbf{P}_A = \mathbf{T}^{-1}\tilde{\mathbf{P}}_A\mathbf{S}$.        (A.16)
7 end
```

Algorithm A.2: The Gold Standard Algorithm.

Suppose all the points $\mathbf{X}_i$ are normalized such that $\mathbf{X}_i = (X_i, Y_i, Z_i, 1)^\top$ and $\mathbf{x}_i = (x_i, y_i, z_i, 1)^\top$, and the last row of $\mathbf{P}$ has the affine form. Then for a single correspondence we get the equation

$$\begin{bmatrix} \mathbf{0}^\top & -\mathbf{X}_i^\top \\ \mathbf{X}_i^\top & \mathbf{0}^\top \end{bmatrix} \begin{pmatrix} \mathbf{P}^1 \\ \mathbf{P}^2 \end{pmatrix} + \begin{pmatrix} y_i \\ -x_i \end{pmatrix} = 0 . \tag{A.17}$$

These equations are stacked up and the system is solved to get an estimate the affine camera. Algorithm A.2 (reproduced from Hartley and Zisserman [52]) gives the Gold Standard Algorithm for estimating an affine camera matrix $\mathbf{P}_A$. Under the assumption of Gaussian measurement errors this algorithm returns the Maximum Likelihood estimate of $\mathbf{P}_A$. Hartley and Zisserman [52] discuss the numerical stability and error analysis of the Gold Standard Algorithm in greater detail.

B

The Exponential Map Parameterization of Rotations

The primary applications of rotations in graphics are to encode orientations and describe and control the motion of rigid bodies and articulations in transformation hierarchies. We use inverse kinematics for posing the skeleton embedded inside the character, and we want our posing algorithm to be efficient and work at interactive rates. For this purpose an appropriate choice of rotation parameterization is essential. Parameterizing rotations is problematic mainly because rotations are non-Euclidean in nature (traveling infinitely far in any direction will bring you back to your starting point an infinite number of times). Any attempt to parameterize the entire set of 3-degree-of-freedom (DOF) rotations by an open subset of Euclidean space (as do Euler angles) will suffer from the *gimbal lock*, i.e., the loss of rotational degrees of freedom, due to singularities[1] in the parameter space. Parameterizations that are themselves defined over non-Euclidean spaces (such as the set of unit quaternions embedded in $\mathbb{R}^4$) may remain singularity-free, and thus avoid the gimbal lock. Employing such parameterizations is complicated, however, since the numerical tools most often employed in graphics assume Euclidean parameterizations; therefore, we must either develop new tools whose domains are non-Euclidean or impose explicit constraints that distinguish the non-Euclidean parameter space from the Euclidean space in which it is embedded (as we must impose constraints that ensure quaternions retain unit length).

Every nonzero vector in $\mathbb{R}^3$ has a direction and magnitude. We can associate a rotation with each vector by specifying the direction as an axis of rotation and the magnitude as the amount by which to rotate around the axis. If we augment this relationship by associating the zero vector with the identity rotation, the relationship is continuous and is known as the exponential map [48]. Unlike the quaternion parameterization, this parameterization is Euclidean, so it contains singularities. In the following sections we present the exponential map in detail and also examine its strengths and limitations as a rotation parameterization.

[1] Intuitively, a singularity is a continuous subspace of the parameter space, all of whose elements correspond to the same rotation; thus, movement within the subspace produces no change in rotation.

B.1 Exponential Maps

The exponential map maps a vector in $\mathbb{R}^3$ describing the axis and magnitude of a 3-DOF rotation to the corresponding rotation. There are many different formulations of the exponential map. There are, however, several advantages to using a map from $\mathbb{R}^3$ to $\mathbb{S}^3$ and using standard quaternion-to-matrix formulae for conversion to $\mathbb{SO}(3)$. Here $\mathbb{R}^3$ is the 3D Euclidean space. $\mathbb{S}^3$ is the underlying set of the subgroup of unit-length quaternions. $\mathbb{SO}(3)$ is the group of all 3×3 matrices whose columns are of unit length and are mutually orthogonal, under the operation of matrix multiplication.

The advantages of not directly mapping to $\mathbb{SO}(3)$ are that the inverse of the exponential map, the log map from $\mathbb{S}^3$ to $\mathbb{R}^3$, is much simpler than the log map from $\mathbb{SO}(3)$ to $\mathbb{R}^3$ and that it is easier to convert to and from $\mathbb{S}^3$ when we need to perform optimal interpolation of rotations using quaternions.

We can formulate an exponential map from $\mathbb{R}^3$ to $\mathbb{S}^3$ as follows:

$$e^{\mathbf{v}} = \begin{cases} [0,0,0,1]^{\mathrm{T}} & \text{if } \mathbf{v} = \mathbf{0}\,, \\ \sum_{m=0}^{\infty}(\frac{1}{2}\bar{\mathbf{v}})^m = [\sin(\frac{1}{2}\theta)\hat{\mathbf{v}}, \cos(\frac{1}{2}\theta)]^{\mathrm{T}} & \text{if } \mathbf{v} \neq \mathbf{0}\,, \end{cases} \tag{B.1}$$

where $\theta = \|\mathbf{v}\|$ and $\hat{\mathbf{v}} = \mathbf{v}/\|\mathbf{v}\|$, which maps $\mathbf{v}$ to a unit quaternion representing a rotation of θ (i.e., $\|\mathbf{v}\|$) about $\mathbf{v}$, where $(\frac{1}{2}\bar{\mathbf{v}})^m$ is computed using quaternion multiplication. The right-hand side of Equation (B.1) is exactly the same as is used to create a unit quaternion from a (unit) axis-angle description of a rotation. The exponential map, however, allows us to encode both the magnitude and axis of rotation into a single 3-vector.

The only problem with this particular formulation is that calculating $\hat{\mathbf{v}} = \mathbf{v}/\|\mathbf{v}\|$, as $\|\mathbf{v}\|$ goes to zero, becomes numerically unstable. We can compute the exponential map robustly in the neighbourhood of the origin by rearranging the above formula. Let

$$\mathbf{q} = e^{\mathbf{v}} = [\sin(\tfrac{1}{2}\theta)\frac{\mathbf{v}}{\theta}, \cos(\tfrac{1}{2}\theta)]^{\mathrm{T}} = [\frac{\sin(\frac{1}{2}\theta)}{\theta}\mathbf{v}, \cos(\tfrac{1}{2}\theta)]^{\mathrm{T}}\,. \tag{B.2}$$

Hence, we reorganize the problematic term so that instead of computing $\mathbf{v}/\|\mathbf{v}\|$ (i.e., $\mathbf{v}/\theta$), we compute $\sin(\frac{1}{2}\theta)/\theta$. This is because $\sin(\frac{1}{2}\theta)/\theta = \frac{1}{2}sinc(\frac{1}{2}\theta)$ and *sinc*, the *sine cardinal* function as given by Equation (B.3), is known to be computable and continuous at and around zero.

$$sinc(\theta) = \begin{cases} 1 & \text{for } \theta = 0\,, \\ \frac{\sin\theta}{\theta} & \text{otherwise}\,. \end{cases} \tag{B.3}$$

Since *sinc* is not included in standard math libraries, we compute it using the Taylor expansion of sin as

$$\begin{aligned} \frac{\sin(\frac{1}{2}\theta)}{\theta} &= \frac{1}{\theta}\left(\frac{\theta}{2} + \frac{(\frac{\theta}{2})^3}{3!} - \frac{(\frac{\theta}{2})^5}{5!} + \cdots\right) \\ &= \frac{1}{2} + \frac{\theta^2}{48} - \frac{\theta^4}{2^5 \cdot 5!} + \cdots \end{aligned} \tag{B.4}$$

Hence, we see that the term is well-defined and that evaluating the entire infinite series would give us the exact value. But as $\theta \to 0$, each successive term is smaller than the last, and terms are alternately added and subtracted; so if we approximate the true value by the first n terms, the error is no greater than the magnitude of the $(n+1)$st term. In fact, since machine precision is limited, we can evaluate the function with no *numerical* error. When $\theta \leq \sqrt[4]{\text{machine precision}}$, use just the first two terms of the expansion

$$\frac{\sin(\frac{1}{2}\theta)}{\theta} = \frac{1}{2} + \frac{\theta^2}{48}\,. \tag{B.5}$$

Otherwise, we perform the actual sin computation and division by θ. Since all the dropped terms involve factors of θ, the approximation and actual function agree at $\theta = 0$.

B.2 Derivatives with Respect to the Exponential Maps

In order to compute the Jacobian of a node in a transformation hierarchy with respect to all of the end effectors below it in the hierarchy, we have to compute the partial derivatives of the rotation matrix. We are, in essence, reparameterizing quaternions; hence, we can compute the derivatives of $\mathbf{R}$ (the rotation matrix) with respect to its exponential map parameters by applying the chain rule. We compute $\partial\mathbf{R}/\partial\mathbf{v}$ as

$$\frac{\partial\mathbf{R}}{\partial\mathbf{v}} = \frac{\partial\mathbf{R}}{\partial\mathbf{q}}\frac{\partial\mathbf{q}}{\partial\mathbf{v}}\,. \tag{B.6}$$

Since we already know how to compute the partial derivatives $\partial\mathbf{R}/\partial\mathbf{q}$, the only new quantities we need are the 12 partial derivatives of the quaternion with respect to its exponential map parameters (i.e., $\partial\mathbf{q}/\partial\mathbf{v}$). To express the similarity in the form of the 12 derivatives, we let l range over the three components of $\mathbf{q}$ that make up its vector part and n range over the components of $\mathbf{v}$. The formulae for computing the partial derivatives of $\mathbf{q}$ with respect to $\mathbf{v}$ are, in the usual case where $\theta >> 0$

$$\begin{aligned}
\frac{\partial q_w}{\partial v_n} &= -\frac{1}{2}v_n\frac{\sin(\frac{1}{2}\theta)}{\theta}\,,\\
\frac{\partial q_l}{\partial v_n} &= \begin{cases} \frac{1}{2}v_n^2\frac{\cos(\frac{1}{2}\theta)}{\theta^2} - v_n^2\frac{\sin(\frac{1}{2}\theta)}{\theta^3} + \frac{\sin(\frac{1}{2}\theta)}{\theta} & \text{if } l = n\,,\\ \frac{1}{2}v_l v_n\frac{\cos(\frac{1}{2}\theta)}{\theta^2} - v_l v_n\frac{\sin(\frac{1}{2}\theta)}{\theta^3} & \text{if } l \neq n\,,\end{cases}
\end{aligned} \tag{B.7}$$

where the quaternion $\mathbf{q}$ is given by $[q_x, q_y, q_z, q_w]^{\mathsf{T}}$. In the neighbourhood of $\theta \to 0$, we can again replace sin and cos by their Taylor series expansions and, after simplifying, discard all terms with powers θ^4 or greater in the numerator. If $\mathrm{TSinc}(\theta) = \frac{1}{2} - \frac{\theta^2}{48}$, then the partial derivatives of q have the form

$$\begin{aligned}
\frac{\partial q_w}{\partial v_n} &= -\frac{1}{2}v_n\ \mathrm{TSinc}(\theta)\,,\\
\frac{\partial q_l}{\partial v_n} &= \begin{cases} \frac{v_n^2}{24}(\frac{\theta^2}{40} - 1) + \mathrm{TSinc}(\theta) & \text{if } l = n\,,\\ \frac{v_l v_n}{24}(\frac{\theta^2}{40} - 1) + \mathrm{TSinc}(\theta) & \text{if } l \neq n\,.\end{cases}
\end{aligned} \tag{B.8}$$

B.3 Strengths and Limitations of the Exponential Map

No single parameterization of rotations is best for all applications (in our own system for view-dependent animation we use the exponential map and Euler angles for inverse kinematics, quaternions for interpolation, and rotation matrices for transformation hierarchies). The exponential map computation of 3- and 2-DOF rotations, however, is very robust and outperforms other parameterizations for inverse kinematics computations. We conclude with a summary of the main strengths and weaknesses of the exponential map.

Strengths:

- The exponential map remains free from gimbal lock over a range of axis-angle rotations up to magnitude 2π, which is suitable for any control or optimization algorithm that operates at single instants of time, provided time marches forward in small steps.
- The exponential map uses three parameters to parameterize $\mathbb{SO}(3)$, which means
  - There is no need for normalization after integrating ordinary differential equations.
  - There is no danger of falling out of a meaningful subspace (like falling off $\mathbb{S}^3$ in $\mathbb{R}^4$), so we do not need explicit constraints.
  - Smaller dimension state vectors combine with the previous point to result in faster performance.
- Interpolation using ordinary cubic splines is possible and may often produce visually acceptable results provided successive key frames are not too distant from each other in $\mathbb{R}^3$.

Limitations:

- There is no simple formula for combining rotations in $\mathbb{R}^3$ akin to quaternion multiplication in $\mathbb{S}^3$ or matrix multiplication in $\mathbb{SO}(3)$.

C

Spherical Joint Limits with Reach Cones

The task of animating humans and animals is greatly aided by automatic constraints, such as joint limits and collision detection, which provide natural restrictions on realistic motion. Such constraints are important in interactive placement, as then the user does not have to recognize unrealistic positions visually. They are essential in more automated methods, such as physical simulation, inverse kinematics, and motion capture from monocular video. Generally, articulated body models used in computer graphics model joints with more than 1-DOF as a sequence of 1-DOF joints specified as Euler angles. The obvious way to specify a range of movement for 1-DOF *hinge joints* is to give a minimum and maximum value in degrees. Joint limits on 2- or 3-DOF joints then become ranges around each single axis. If the longitudinal axis of a segment distal to the joint is the z axis, a 2-DOF *universal joint* allows rotation about the x and y axes. If the joint also allows longitudinal z axis rotation, it is a 3-DOF *ball-and-socket joint*. Single-axis range specification is inadequate for such joints.

A more natural specification, sometimes called a *joint sinus cone*, has been used in biomechanics, simulation, and computer graphics. Here the range of motion is described as an irregular cone, defined by a cyclical sequence of points on the sphere that represents free universal movement. Wilhelms and Van Gelder [137] refer to such limits as *reach cones*. They describe a new method for specifying, recognizing inclusion in, and intersecting reach cones. We have used their techniques for specifying and enforcing joint limits in inverse kinematics (see Section 3.5.1).

C.1 Defining Reach Cones

A *joint* is an articulation between a *parent segment* and *child segment* of a tree structured articulated body. The end of the segment closest to the root of the tree is called *proximal* and the other end is called *distal*. The parent segment is said to be proximal to the joint and the child segment is said to be distal. When not otherwise qualified, the term segment should be understood to mean the segment distal to the joint under discussion, i.e., the child segment. Each segment's longitudinal axis is considered to be a (bounded) straight line segment originating at the joint with its parent segment. Henceforth, we refer to the segment's longitudinal axis as the *longitudinal segment axis* or as the longitudinal axis.

Formally, a *cone* is a set of rays starting at the origin. These rays can be defined by set of points on a sphere centered at the origin. Without any joint limit, the distal end of the segment might be anywhere on this sphere. With joint limits, the set of points that can actually be occupied by the distal end represents, or defines, the *sinus* or *reach cone*. Alternatively, the intersection of the sinus cone with this sphere defines the set of points that can be occupied by the distal end of the segment. Spherical joint limits are specified as a reach cone inscribed on a sphere of radius one, centered at the joint, together with limits on rotation about the longitudinal axis, that may vary throughout the reach cone. The reach cone is specified by a spherical polygon, called the *reach cone polygon*. The vertices of this spherical polygon are a series of *boundary points* on this unit sphere, and great-circle arcs on the sphere form its edges. Points inside or on the reach-cone polygon are considered to be within the reach cone. Reach cones are defined in the default coordinate frame of the segment distal to the joint. They do not move about with state rotations of this segment. Detection of allowable positions in the reach cone is done by testing whether the longitudinal segment axis intersects the unit sphere inside or outside the reach cone polygon. At each vertex, limits on rotation about the longitudinal segment axis (also called *twist*) may be specified by maximum and minimum angles. Limits on the twist at any position in the reach cone are defined by an interpolant of these values. This capability is important because it has been found that the range of motion for longitudinal rotations is a function of the direction of the longitudinal axis [134].

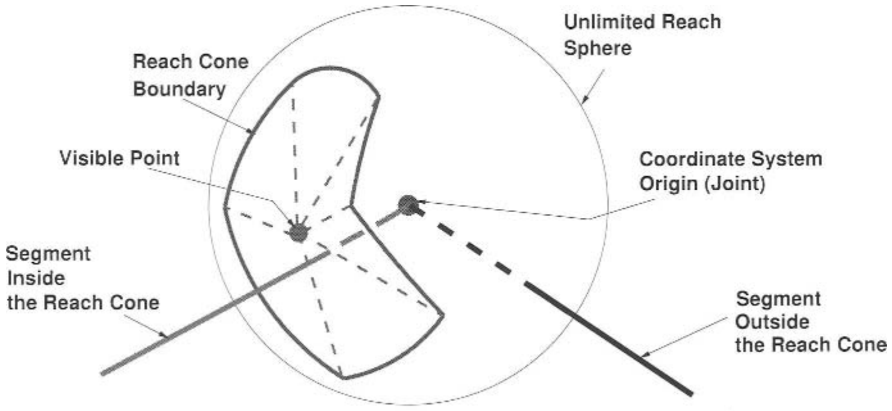

Unlimited Reach Sphere
Reach Cone Boundary
Visible Point
Coordinate System Origin (Joint)
Segment Inside the Reach Cone
Segment Outside the Reach Cone

Fig. C.1. A reach cone polygon with five boundary points and a visible point.

It is mandatory for the reach cone polygon to have a *visible point*, that is, a point that can be "seen" by all of the boundary points in the sense that a great-circle arc (or line segment) joining the boundary point with the visible point lies entirely inside the reach cone (see Fig. C.1).

C.2 Detecting Reach Cone Inclusion

Suppose a reach cone polygon has been defined with origin O, visible point V, and boundary points P_i for $i = 0, \dots, n-1$. The points are treated as 3D vectors from the origin, and arithmetic on indexes is understood to be modulo n.

Observe that for each i, the four points (O, V, P_i, P_{i+1}) define a tetrahedron. The order of the points imparts an orientation to the tetrahedron. These n tetrahedra also generate the reach cone and so can be used as an alternative representation. The volume of an oriented tetrahedron with one point at the origin is given by the triple scalar product expression as

$$\mathrm{vol}(O, a, b, c) = \frac{1}{6}\, a \times b \cdot c \,. \tag{C.1}$$

The visible point is properly positioned with respect to the boundary points if and only if each of the n oriented tetrahedra has positive volume; i.e.,

$$V \times P_i \cdot P_{i+1} > 0 \qquad \text{for} \quad 0 \le i \le n-1 \,. \tag{C.2}$$

We consider the problem of deciding whether a specified vector L is in the reach cone. The vector L is the vector along the segment whose inclusion is being tested. L is in the reach cone if and only if L passes through one of the n tetrahedra that define the reach cone. Also, L passes through the tetrahedron (O, V, P_i, P_{i+1}) if and only if each of the three oriented tetrahedra, (O, V, P_i, L), (O, P_i, P_{i+1}, L), and (O, P_{i+1}, V, L) has nonnegative volume, using Equation (C.1).

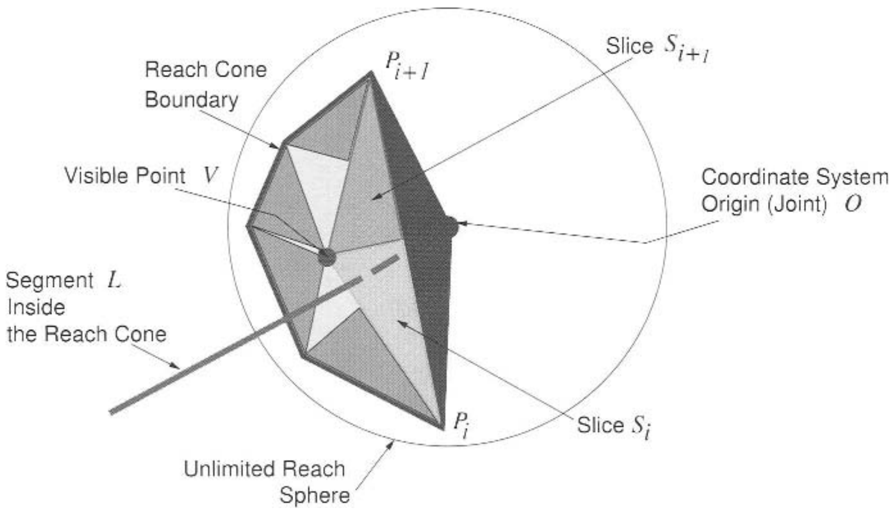


Fig. C.2. A reach cone with five boundary points. Slice planes S_i and S_{i+1} and the boundary plane defined by O, P_i, P_{i+1} enclose the longitudinal segment axis vector L.

Another way to view this is to consider the plane defined by O, V, P_i, which has a normal vector (not necessarily unit length) $V \times P_i$. If L is on the same side of the plane as this normal vector points, then $V \times P_i \cdot L = 0$. The plane defined by O, V, P_i is called a *radial slice plane* and is denoted by S_i (see Fig. C.2). Similarly, for L to pass through the tetrahedron (O, V, P_i, P_{i+1}) it is necessary that $V \times P_{i+1} \cdot L = 0$ and $P_i \times P_{i+1} \cdot L = 0$. Note that the cross products depend on the boundary points and visible point, but not on L, so they can be computed just once when the reach cone is specified:

$$\begin{aligned} S_i &= V \times P_i \,, \\ B_i &= P_i \times P_{i+1} \,. \end{aligned} \tag{C.3}$$

The algorithm to decide whether L is in the reach cone is summarized in Algorithm C.1.

```
Require: Given a reach cone with origin $O$, visible point $V$, and boundary
         points $P_i$ for $i = 0, \ldots, n-1$. The segment to be tested for inclusion
         is given as a vector $L$.
1 begin
2     Find the $i$ such that $p_i = S_i \cdot L \geq 0$ and $p_{i+1} = S_{i+1} \cdot L < 0$. There is
      exactly one such $i$ because the radial slices (as half planes) partition the
      sphere. We start with the slice in which the axis was located previously
      and search for the required $i$. In the worst case, this step takes $n$
      dot-product operations, since the cross products are stored. We can speed
      this up by doing a binary search on the radial slice planes.
3     If $v_i = B_i \cdot L \geq 0$, then $L$ is in the reach cone; otherwise, it is not.
4 end
```

Algorithm C.1: Reach cone inclusion testing algorithm.

C.3 Calculating a Boundary Position

When the segment moves from a valid position in reach cone, to a position outside the reach cone it has to be restricted to a boundary position in order to enforce the joint limits. Hence we need to calculate this boundary position where the segment exits the reach cone. Such a boundary position can be calculated if we assume that the segment moves directly from the current valid position to the new invalid position in a straight-line path.

Let the old position be L_0 and the new position be L. The exit point will be found on the straight line joining L_0 and L. The line will be tested against intersection with the reach cone boundary planes. The boundary plane B_i is computed using Equation (C.3). The line containing L_0 and L is defined as

$$L(t) = L_0 + t(L - L_0)\,. \tag{C.4}$$

The value of t where this line intersects the boundary plane B_i is given by

$$t_i = \frac{-L_0 \cdot B_i}{(L - L_0) \cdot B_i}\,. \tag{C.5}$$

If $0 \leq t \leq 1$, an intersection has occurred. Further, we need to check that the intersection point lies between the slice planes S_i and S_{i+1}, where slice plane S_i is defined as

$$S_i = V_i \times P_{i+1}\,. \tag{C.6}$$

C.4 Twist Limits

For a ball-and-socket joint the limits of rotation about the longitudinal depends on the direction of the segment. For example, in the case of the human shoulder it has been found that the limits of rotation of the upper arm vary considerably between 94 degrees and 157 degrees depending on the arm orientation. Therefore,the longitudinal rotation limits are specified for each boundary point and visible point. These values are then interpolated to find the twist limits at any valid position inside the reach cone.

Given the minimum and maximum twist rotation limits at each of the reach cone points including the visible point, the limits at any segment orientation L can be computed as follows. Find i such that $p_i = S_i \cdot L0$, $p_{i+1} = S_{i+1} \cdot L < 0$, and $v_i = B_i \cdot L \geq 0$, where S_i and B_i are computed using Equation (C.3). Now we define an averaging factor s as $s = p_i + p_{i+1} + vi$ and weights of individual points of the spherical polygonal in which segment L lies as $w_i = p_i/s$, $w_i = p_{i+1}/s$, and $w_v = v_i/s$. The limits at any segment L are then computed as

$$\begin{aligned}\theta_{min}(L) &= w_i\theta_{min}(P_i) + w_{i+1}\theta_{min}(P_{i+1}) + w_v\theta_{min}(V)\,,\\ \theta_{max}(L) &= w_i\theta_{max}(P_i) + w_{i+1}\theta_{max}(P_{i+1}) + w_v\theta_{max}(V)\,.\end{aligned} \tag{C.7}$$

C.5 Cyclic Order of Boundary Points

Reach cones must be defined such that the boundary points are in counterclockwise order when viewed from the outside and above the visible point. This property is used in inclusion testing, computation of boundary position, and twist limits at any intermediate position inside the reach cone. The user can take care of the order of the points while specifying them interactively. It is also possible to compute the order automatically. The visible point is of great help in finding out automatic order. By the definition of visible point, it is a point that can be seen by all the points directly.

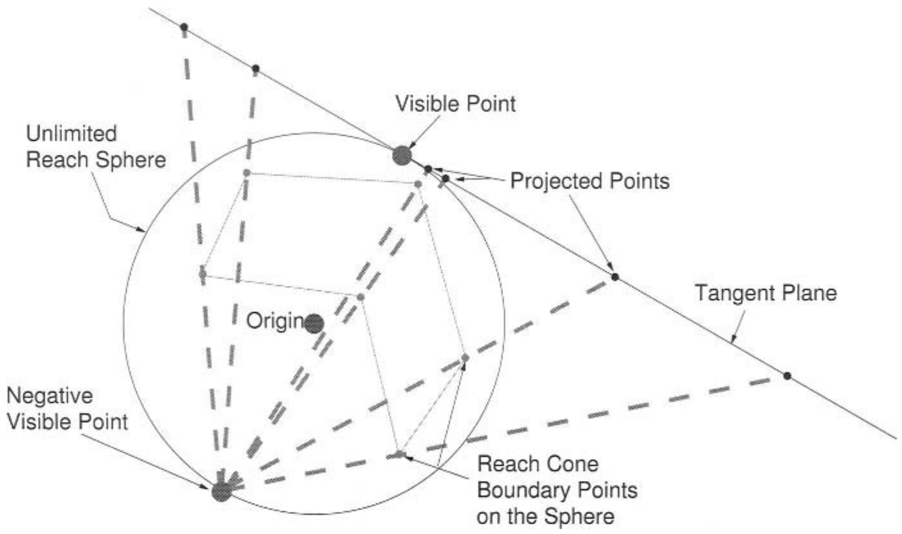


Fig. C.3. The stereographic projection provides an invertible mapping between the entire sphere (except for one point) and the tangent plane.

Define a plane to the sphere at visible point and project the boundary points on to this plane through the negative of visible point (see Fig. C.3). Now the order of the points in the reach cone is the same as that on the plane. Order of the points can be found by computing the angle each of the points make with a reference line passing through the visible point. The angles thus obtained are sorted to compute the automatic order of boundary points.

C.6 Interactively Creating the Reach Cone

Boundary points for a reach cone can be defined interactively or from a file. We have created a GUI in which points are added and can be repositioned. The GUI designed has the functionality to add, delete, and specify the order of the points to create a reach cone. User is provided with the facility to compute the visible point automatically by simply finding the resultant vector of all the normalized boundary point vectors. The user may reposition the visible point if the automatically computed point is not meeting the necessary condition for the location of the visible point. For some joints like the wrist joint in humans, the reach cone can be approximated by an ellipse. Detailed description of the interface use and implementation can be found in [70]. Reach cones created for the Hugo mesh are shown in Fig. C.4.

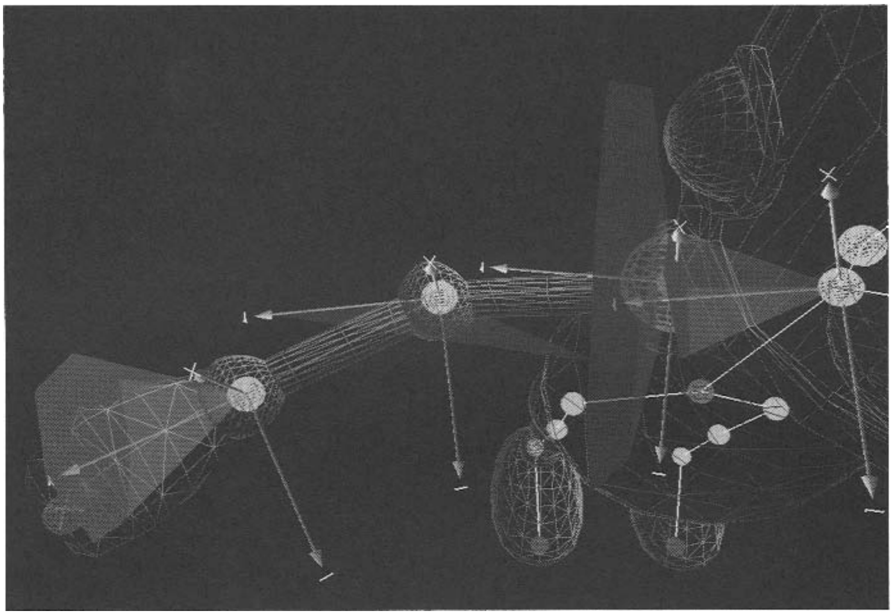


Fig. C.4. Reach cones for Hugo's arm. The reach cone at the elbow allows only 1 degree of freedom, the one at the wrist joint allows 2, and the one at the shoulder joint allows 3 degrees of freedom.

References

1. A. Adamson and V. Jenson. *Shrek.* DreamWorks Animation SKG, 2001.
2. A. Agarwala. SnakeToonz: A semi-automatic approach to creating cel animation from video. In *Proceedings of the 2nd International Symposium on Non-Photorealistic Animation and Rendering*. ACM Press, 2002.
3. A. Agarwala, A. Hertzmann, D. H. Salesin, and S. M. Seitz. Keyframe-based tracking for rotoscoping and animation. *ACM Transactions on Graphics*, 23(3):584–591, 2004.
4. M. Agrawala, D. Zorin, and T. Munzner. Artistic multiprojection rendering. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pages 125–136, 2000.
5. J. Amat, A. Casals, and M. Frigola. Stereoscopic system for human body tracking in natural scenes. In *Proceedings of ICCV Workshop on Modeling People*, Sept. 1999.
6. D. Arijon. *Grammar of the Film Language.* Communication Arts books, Hastings House, New York, 1976.
7. O. Arikan and D. A. Forsyth. Interactive motion generation from examples. In *SIGGRAPH '02: Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, pages 483–490. ACM Press, 2002.
8. O. Arikan, D. A. Forsyth, and J. F. O'Brien. Motion synthesis from annotations. *ACM Transactions on Graphics*, 22(3):402–408, 2003.
9. K. Asbury and L. Cook. *Spirit: Stallion of the Cimarron.* DreamWorks SKG, 2002.
10. S. Avidan and A. Shashua. Threading fundamental matrices. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(1):73–77, Jan. 2001.
11. R. Azuma. Tracking requirements for augmented reality. *Communications of the ACM*, 36(7):50–51, July 1993.
12. N. I. Badler, K. H. Manoochehri, and G. Walters. Articulated figure positioning by multiple constraints. *IEEE Computer Graphics and Applications*, 7(6):28–38, June 1987.
13. P. Baerlocher. *Inverse kinematics techniques for the interactive posture control of articulated figures.* PhD thesis, EPFL, 2001.
14. P. Beardsley, A. Zisserman, and D. Murray. Sequential updating of projective and affine structure from motion. *International Jounral of Computer Vision*, 23:235–259, 1997.
15. B. Bird. *The Incredibles.* Pixar Animation Studios and Walt Disney Pictures, 2004.
16. A. Blake and M. Isard. *Active Contours: The Application of Techniques from Graphics, Vision, Control Theory, and Statistics to Visual Tracking of Shapes in Motion.* Springer-Verlag, New York, 1999.

17. L. Boissieux. Hugo (3D mesh model). *Eurographics 2004 mascot, INRIA Rhône-Alpes*, 2004.
18. Boujou. 2d3 Ltd. 2002. http://www.2d3.com.
19. M. Brand and A. Hertzmann. Style Machines. In *SIGGRAPH '00: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pages 183–192. ACM Press/Addison-Wesley Publishing Co., 2000.
20. C. Bregler. Learning and recognizing human dynamics in video sequences. In *CVPR '97: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* IEEE Computer Society, 1997.
21. C. Bregler, M. Covell, and M. Slaney. Video Rewrite: Driving visual speech with audio. In *SIGGRAPH '97: Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, pages 353–360. ACM Press/Addison-Wesley Publishing Co., 1997.
22. C. Bregler, L. Loeb, E. Chuang, and H. Deshpande. Turning to the masters: Motion capturing cartoons. In *SIGGRAPH '02: Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, pages 399–407. ACM Press, 2002.
23. C. Bregler and J. Malik. Tracking people with twists and exponential maps. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* IEEE Computer Society, 1998.
24. S. Capell, S. Green, B. Curless, T. Duchamp, and Z. Popović. Interactive skeleton-driven dynamic deformations. *ACM Transactions on Graphics*, 21(3):586–593, July 2002.
25. P. Chaudhuri, P. Kalra, and S. Banerjee. A system for view-dependent animation. *Computer Graphics Forum*, 23(3):411–420, 2004.
26. J. J. Cherlin, F. Samavati, M. C. Sousa, and J. A. Jorge. Sketch-based modeling with few strokes. In *21st Spring Conference on Computer Graphics*, 2005.
27. C. Christensen and S. Corneliussen. Visualization of human motion using model-based vision. Technical report, Laboratory of Image Analysis, Aalborg University, Denmark, Jan. 1997.
28. J. M. Cohen, J. F. Hughes, and R. C. Zeleznik. Harold: A world made of drawings. In *Proceedings of the 1st International Symposium on Non-Photorealistic Animation and Rendering*, pages 83–90. ACM Press, 2000.
29. P. Coleman and K. Singh. Ryan: Rendering your animation nonlinearly projected. In *Proceedings of the 3rd International Symposium on Non-Photorealistic Animation and Rendering*, pages 129–156. ACM Press, 2004.
30. J. Collomosse, D. Rowntree, and P. Hall. Cartoon-style rendering of motion from video. In *Proceedings of Vision, Video and Graphics*, pages 117–124, July 2003.
31. D. Cooper. 2D/3D hybrid character animation on *spirit.* In *SIGGRAPH 2002: Sketches and Applications*, 2002.
32. K. Cornelis, M. Pollefeys, M. Vergauwen, F. Verbiest, and L. V. Gool. Tracking based structure and motion recovery for augmented video productions. In *Proceedings of the ACM Symposium on Virtual Reality and Software Technology (VRST) 2001*, pages 17–24, 2001.
33. D. Cornish, A. Rowan, and D. Luebke. View-dependent particles for interactive non-photorealistic rendering. In *Proceedings of Graphics Interface 2001*, pages 151–158, June 2001.
34. E. Cosatto and P. Graf. Photo-realistic talking heads from image samples. *IEEE Transaction on Multimedia*, 2(3):152–163, 2000.
35. E. Darnell and T. McGrath. *Madagascar.* DreamWorks Animation SKG, 2005.

36. J. Davis, M. Agrawala, E. Chuang, Z. Popović, and D. Salesin. A sketching interface for articulated figure animation. In *Proceedings of ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 320–328. Eurographics Association, 2003.
37. P. Debevec, G. Borshukov, and Y. Yu. Efficient view-dependent image-based rendering with projective texture-mapping. In *Proceedings of the 9th Eurographics Rendering Workshop*, June 1998.
38. D. DeCarlo and A. Santella. Stylization and abstraction of photographs. In *SIGGRAPH '02: Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, pages 769–776. ACM Press, 2002.
39. J. Deutscher, B. North, B. Bascle, and A. Blake. Tracking through singularities and discontinuities by random sampling. In *Proceedings of International Conference on Computer Vision 1999*, pages 1144–1149, 1999.
40. A. Fitzgibbon and A. Zisserman. Automatic camera recovery for closed or open image sequences. In *Proceedings of European Conference on Computer Vision*, pages 311–326, 1998.
41. A. Fitzgibbon and A. Zisserman. Automatic camera tracking. In M. Shah and R. Kumar, editors, *Video Registration*, pages 18–35. Kluwer Academic, 2003.
42. J. Funge, X. Tu, and D. Terzopoulos. Cognitive modeling: Knowledge, reasoning and planning for intelligent characters. In *SIGGRAPH '99: Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, pages 29–38. ACM Press/Addison-Wesley Publishing Co., 1999.
43. A. S. Glassner. Cubism and cameras: Free-form optics for computer graphics. *Technical Report (MSR-TR-2000-05)*, Jan. 2000.
44. M. Gleicher and N. Ferrier. Evaluating video-based motion capture. In *Proceedings of the Computer Animation*, pages 75–80. IEEE Computer Society, 2002.
45. M. Gleicher and A. Witkin. Through-the-lens camera control. In *SIGGRAPH '92: Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, pages 331–340. ACM Press, 1992.
46. O. Goemans and M. Overmars. Automatic generation of camera motion to track a moving guide. Technical Report, Institute of Information and Computer Sciences, University of Utrecht, July 2004.
47. G. H. Golub and C. F. V. Loan. *Matrix Computations, 3rd edition.* John Hopkins University Press, Baltimore, 1996.
48. F. S. Grassia. Practical parameterization of rotations using the exponential map. *Journal of Graphics Tools*, 3(3):29–48, 1998.
49. K. Grochow, S. L. Martin, A. Hertzmann, and Z. Popović. Style-based inverse kinematics. *ACM Transactions on Graphics*, 23(3):522–531, 2004.
50. Y. Guo, G. Xu, and S. Tsuji. Tracking human body motion based on a stick-figure model. *Journal of Visual Communication and Image Representation*, 5:1–9, 1994.
51. A. Haro and I. Essa. Learning video processing by example. In *Proceedings of the 16th International Conference on Pattern Recognition*, volume 1, pages 487–491, 2002.
52. R. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision.* Cambridge University Press, 2000.
53. J. Hays and I. Essa. Image and video based painterly animation. In *Proceedings of the 3rd International Symposium on Non-Photorealistic Animation and Rendering*, pages 113–120. ACM Press, 2004.
54. L. He, M. F. Cohen, and D. H. Salesin. The virtual cinematographer: A paradigm for automatic real-time camera control and directing. In *SIGGRAPH '96: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, pages 217–224. ACM Press, 1996.

55. R. Hecker and K. Perlin. Controlling 3D objects by sketching 2D views. In *SPIE - Sensor Fusion V*, volume 1828, pages 46–48, Nov. 1992.
56. A. Hertzmann. Painterly rendering with curved brush strokes of multiple sizes. In *SIGGRAPH '98: Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, pages 453–460. ACM Press, 1998.
57. A. Hertzmann. Fast paint texture. In *Proceedings of the 2nd International Symposium on Non-Photorealistic Animation and Rendering*, pages 91–96. ACM Press, 2002.
58. A. Hertzmann, N. Oliver, B. Curless, and D. Salesin. Image analogies. In *SIGGRAPH '01: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, pages 327–340. ACM Press, 2001.
59. A. Hertzmann and K. Perlin. Painterly rendering for video and interaction. In *Proceedings of the 1st International Symposium on Non-Photorealistic Animation and Rendering*, pages 7–12. ACM Press, 2000.
60. D. C. Hogg. Model-based vision: A program to see a walking person. *Image and Vision Computing*, 1(1):5–20, 1983.
61. D. C. Hogg. *Interpreting images of a known moving object.* PhD thesis, University of Sussex, U.K., 1984.
62. H. Hoppe. View-dependent refinement of progressive meshes. In *SIGGRAPH '97: Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, pages 189–198. ACM Press/Addison-Wesley Publishing Co., 1997.
63. E. Hsu, K. Pulli, and J. Popović. Style translation for human motion. In *SIGGRAPH '05: Proceedings of the 32rd Annual Conference on Computer Graphics and Interactive Techniques*, 2005.
64. W. M. Hsu, J. F. Hughes, and H. Kaufman. Direct manipulation of free-form deformations. In *SIGGRAPH '92: Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, pages 177–184. ACM Press, 1992.
65. T. Igarashi and J. F. Hughes. Smooth meshes for sketch-based freeform modeling. In *SI3D '03: Proceedings of the 2003 Symposium on Interactive 3D Graphics*, pages 139–142. ACM Press, 2003.
66. T. Igarashi, S. Matsuoka, and H. Tanaka. Teddy: A sketching interface for 3D freeform design. In *SIGGRAPH '99: Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, pages 409–416. ACM Press/Addison-Wesley Publishing Co., 1999.
67. T. Igarashi, T. Moscovich, and J. F. Hughes. Spatial keyframing for performance-driven animation. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 107–115. ACM Press, 2005.
68. M. Isard and A. Blake. CONDENSATION — conditional density propagation for visual tracking. *International Journal of Computer Vision*, 29(1):5–28, 1998.
69. Y. Iwai, K. Ogaki, and M. Yachida. Posture estimation using structure and motion models. In *Proceedings of the International Conference on Computer Vision*, pages 214–219, 1999.
70. A. Jindal. *Interactive tools for IK, deformation and tracking.* Master's thesis, Department of Computer Science and Engineering, Indian Institute of Technology Delhi, 2004.
71. N. Jojic, J. Gu, H. C. Shen, and T. S. Huang. 3-D reconstruction of multipart self-occluding objects. In *ACCV '98: Proceedings of the Third Asian Conference on Computer Vision-Volume II*, pages 455–462. Springer-Verlag, 1998.
72. I. Kakadiaris and D. Metaxas. Vision-based animation of digital humans. In *CA '98: Proceedings of the Computer Animation*, page 144. IEEE Computer Society, 1998.
73. R. E. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

74. R. D. Kalnins, L. Markosian, B. J. Meier, M. A. Kowalski, J. C. Lee, P. L. Davidson, M. Webb, J. F. Hughes, and A. Finkelstein. WYSIWYG NPR: Drawing strokes directly on 3D models. *ACM Transactions on Graphics*, 21(3):755–762, July 2002.
75. Y. Kameda, M. Minoh, and K. Ikeda. Three dimensional pose estimation of an articulated object from its silhouette image. In *Proceedings of the Asian Conference on Computer Vision*, 1993.
76. R. J. Kate, P. Kalra, and S. Banerjee. Towards an automatic approach for view-dependent geometry. *International Journal of Image and Graphics (IJIG)*, 2(3):413–423, 2002.
77. L. Kavan and J. Zára. Real-time skin deformation with bones blending. *WSCG Short Papers proceedings*, 2003.
78. Y. Kho and M. Garland. Sketching mesh deformations. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, pages 147–154. ACM Press, 2005.
79. A. W. Klein, P.-P. J. Sloan, A. Finkelstein, and M. F. Cohen. Stylized video cubes. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 15–22. ACM Press, 2002.
80. K. G. Kobayashi and K. Ootsubo. t-FFD: free-form deformation by using triangular mesh. In *Proceedings of the 8th ACM Symposium on Solid modeling and applications*, pages 226–234. ACM Press, 2003.
81. L. Kovar, M. Gleicher, and F. Pighin. Motion graphs. In *SIGGRAPH '02: Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, pages 473–482. ACM Press, 2002.
82. M. A. Kowalski, L. Markosian, J. D. Northrup, L. Bourdev, R. Barzel, L. S. Holden, and J. F. Hughes. Art-based rendering of fur, grass, and trees. In *SIGGRAPH '99: Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, pages 433–438. ACM Press/Addison-Wesley Publishing Co., 1999.
83. J. Lander. Skin them bones: Game programming for the web generation. *Game Developer Magazine*, pages 11–16, May 1998.
84. J. Lander. Over my dead, polygonal body. *Game Developer Magazine*, pages 17–22, Oct. 1999.
85. C. Landreth. *Ryan*. Copper Heart Entertainment Inc. and National Film Board of Canada, 2004.
86. J. Lasseter. Principles of traditional animation applied to 3D computer animation. In *SIGGRAPH '87: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, pages 35–44. ACM Press, 1987.
87. H. J. Lee and Z. Chen. Determination of 3D human body postures from a single view. *Computer Vision, Graphics and Image Processing*, 30(2):148–168, May 1985.
88. J. Lee, J. Chai, P. S. A. Reitsma, J. K. Hodgins, and N. S. Pollard. Interactive control of avatars animated with human motion data. In *SIGGRAPH '02: Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, pages 491–500. ACM Press, 2002.
89. K. H. Lee, M. G. Choi, and J. Lee. Motion patches: Building blocks for virtual environments annotated with motion data. In *SIGGRAPH '06: Proceedings of the 33rd Annual Conference on Computer Graphics and Interactive Techniques*, 2006.
90. J. P. Lewis, M. Cordner, and N. Fong. Pose space deformation: A unified approach to shape interpolation and skeleton-driven deformation. In *SIGGRAPH '00: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pages 165–172. ACM Press/Addison-Wesley Publishing Co., 2000.
91. Y. Li, M. Gleicher, Y.-Q. Xu, and H.-Y. Shum. Stylizing motion with drawings. In *Proceedings of ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 309–319. Eurographics Association, 2003.

92. Y. Li, T. Wang, and H.-Y. Shum. Motion texture: A two-level statistical model for character motion synthesis. In *SIGGRAPH '02: Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, pages 465–472. ACM Press, 2002.
93. S. E. Librande. *Example-based character drawing.* Master's thesis, Massachusetts Institute of Technology, Master of Science in Visual Studies, Sept. 1992.
94. P. Lindstrom and V. Pascucci. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):239–254, 2002.
95. P. Litwinowicz. Processing images and video for an impressionist effect. In *SIGGRAPH '97: Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, pages 407–414. ACM Press/Addison-Wesley Publishing Co., 1997.
96. P. C. Litwinowicz. Inkwell: A 2 -D animation system. In *SIGGRAPH '91: Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, pages 113–122. ACM Press, 1991.
97. C. Mao, S. F. Qin, and D. K. Wright. A sketch-based gesture interface for rough 3D stick figure animation. In *Proceedings of the Eurographics Workshop on Sketch Based Interfaces and Modeling*, pages 175–183. Eurographics Association, 2005.
98. L. Markosian, J. Cohen, T. Crulli, and J. Hughes. Skin: a constructive approach to modeling free-form shapes. In *SIGGRAPH '99: Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, pages 393–400. ACM Press/Addison-Wesley Publishing Co., 1999.
99. D. Martín, S. García, and J. C. Torres. Observer-dependent deformations in illustration. In *Proceedings of the 1st International Symposium on Non-Photorealistic Animation and Rendering*, pages 75–82. ACM Press, 2000.
100. B. J. Meier. Painterly rendering for animation. In *SIGGRAPH '96: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, pages 477–484. ACM Press, 1996.
101. T. B. Moeslund and E. Granum. 3D human pose estimation using 2D-data and an alternative phase space representation. In *Proceedings of Workshop on Human Modeling, Analysis and Synthesis at CVPR*, June 2000.
102. T. B. Moeslund and E. Granum. A survey of computer vision-based human motion capture. *Computer Vision and Image Understanding*, 81(3):231–268, 2001.
103. E. Muybridge. *The Human Figure in Motion.* Dover Publications, New York, 1955.
104. T. Ngo, D. Cutrell, J. Dana, B. Donald, L. Loeb, and S. Zhu. Accessible animation and customizable graphics via simplicial configuration modeling. In *SIGGRAPH '00: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pages 403–410. ACM Press/Addison-Wesley Publishing Co., 2000.
105. M. Owen and P. Willis. Modelling and interpolating cartoon characters. In *Proceedings of Computer Animation '94*, pages 148–155, May 1994.
106. F. J. Perales and J. Torres. A system for human motion matching between synthetic and real images based on a biomechanic graphical model. In *Proceeding of Workshop on Motion of Non-Rigid and Articulated Objects*, pages 83–88, Austin, Texas, 1994.
107. B.-T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, June 1975.
108. R. Plankers, P. Fua, and N. D. Apuzzo. Automated body modeling from video sequences. In *Proceedings of ICCV Workshop on Modeling People*, Sept. 1999.
109. P. Rademacher. View-dependent geometry. In *SIGGRAPH '99: Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, pages 439–446. ACM Press/Addison-Wesley Publishing Co., 1999.

110. J. Rittscher and A. Blake. Classification of human body motion. In *Proceedings of the International Conference on Computer Vision*, pages 634–639, 1999.
111. K. Rohr. Recognition of human movements based on explicit motion models. In *Motion-Based Recognition*, pages 171–198. Kluwer Academic Press, 1997.
112. C. Rose, M. F. Cohen, and B. Bodenheimer. Verbs and adverbs: Multidimensional motion interpolation. *IEEE Computer Graphics and Applications*, 18(5):32–40, 1998.
113. W. R. Sabiston. *Extracting 3D Motion from Hand-Drawn Animated Figures*. Master's thesis, Massachusetts Institute of Technology, Master of Science in Visual Studies, June 1991.
114. H. Sakaguchi and M. Sakakibara. *Final Fantasy: The Spirits Within*. Square Pictures, 2001.
115. D. Schaub. The Polar Express Diary: Part 2 — Performance Capture & the MoCap/Anim Process. *VFX World*, Feb. 2005. http://www.vfxworld.com/?sa=adv &code=319b255d &atype=articles &id=2390.
116. A. Schödl and I. A. Essa. Controlled animation of video sprites. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 121–127. ACM Press, 2002.
117. H. Sidenbladh, F. D. la Torre, and M. J. Black. A framework for modeling the appearance of 3D articulated figures. In *Proceedings of the 4th International Conference on Automatic Face and Gesture Recognition*, pages 368–375, Mar. 2000.
118. G. Simon, A. Fitzgibbon, and A. Zisserman. Markerless tracking using planar structures in the scene. In *Proceedings of International Symposium on Augmented Reality*, pages 120–128, Oct. 2000.
119. K. Singh. A fresh perspective. In *Proceedings of Graphics Interface 2002*, pages 17–24, 2002.
120. J. Starck, G. Miller, and A. Hilton. Video-based character animation. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 49–58. ACM Press, 2005.
121. R. W. Sumner, M. Zwicker, C. Gotsman, and J. Popović. Mesh-based inverse kinematics. *ACM Transactions Graphics*, 24(3):488–495, 2005.
122. D. Sýkora, J. Buriánek, and J. Žára. Sketching cartoons by example. In *Proceedings of the Eurographics Workshop on Sketch Based Interfaces and Modeling*, pages 27–33. Eurographics Association, 2005.
123. C. J. Taylor. Reconstruction of articulated objects from point correspondences in a single uncalibrated image. *Computer Vision and Image Understanding*, 80(3):349–363, 2000.
124. F. Thomas and O. Johnson. *Disney Animation: The Illusion of Life*. Abbeville Press, New York, 1984.
125. M. Thorne, D. Burke, and M. van de Panne. Motion doodles: An interface for sketching character motion. *ACM Transactions on Graphics*, 23(3):424–431, 2004.
126. A. N. Tikhonov. Solution of incorrectly formulated problems and the regularization method. *Soviet Math. Dokl.*, 4:1036–1038, 1963.
127. O. Tolba, J. Dorsey, and L. McMillan. A projective drawing system. In *SI3D '01: Proceedings of the 2001 Symposium on Interactive 3D Graphics*, pages 25–34. ACM Press, 2001.
128. B. Triggs, P. McLauchlan, R. Hartley, and A. Fitzgibbon. Bundle adjustment – a modern synthesis. In *Proceedings of the International Workshop on Vision Algorithms: Theory and Practice*, pages 298–372, 1999.
129. G. Trousdale and K. Wise. *Beauty and the Beast*. Walt Disney Pictures, 1991.
130. R. Ženka and P. Slavík. New dimension for sketches. In *SCCG '03: Proceedings of the 19th Spring Conference on Computer Graphics*, pages 157–163. ACM Press, 2003.

131. L. Wachowski and A. Wachowski. *The Matrix*. Warner Studios, 1999.
132. J. Wang, Y. Xu, H.-Y. Shum, and M. F. Cohen. Video Tooning. *ACM Transactions on Graphics*, 23(3):574–583, 2004.
133. L. Wang, X. Wang, X. Tong, S. Lin, S. Hu, B. Guo, and H.-Y. Shum. View-dependent displacement mapping. *ACM Transactions on Graphics*, 22(3):334–339, 2003.
134. X. Wang, M. Maurin, F. Mazet, N. D. C. Maia, K. Voinot, J. P. Verriest, and M. Fayet. Three-dimensional modelling of the motion range of axial rotation of the upper arm. *Journal of Biomechanics*, 31:899–908, 1998.
135. G. Welch and E. Foxlin. Motion tracking: No silver bullet, but a respectable arsenal. *IEEE Computer Graphics and Applications*, 22(6):24–38, 2002.
136. C. Welman. *Inverse kinematics and geometric constraints for articulated figure manipulation*. Master's thesis, Simon Fraser University, 1993.
137. J. Wilhelms and A. V. Gelder. Fast and easy reach-cone joint limits. *Journal of Graphics Tools*, 6(2):27–41, 2001.
138. H. Winnemoller, S. C. Olsen, and B. Gooch. Real-time video abstraction. In *SIGGRAPH '06: Proceedings of the 33rd Annual Conference on Computer Graphics and Interactive Techniques*, 2006.
139. C. R. Wren, A. Azarbayejani, T. Darrell, and A. Pentland. Pfinder: Real-time tracking of the human body. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(7):780–785, 1997.
140. C. R. Wren and A. P. Pentland. Dynaman: Recursive modeling of human motion. In *Vismod*, 1997.
141. M. Yamamoto and K. Koshikawa. Human motion analysis based on a robot arm model. In *Proceedings of IEEE Computer Vision and Pattern Recognition (CVPR) 1991*, pages 664–665. IEEE, June 1991.
142. Y. Yang, J. X. Chen, and M. Beheshti. Nonlinear perspective projections and magic lenses: 3D view deformation. *IEEE Computer Graphics and Applications*, 25(1):76–84, Jan. 2005.
143. R. C. Zeleznik, K. P. Herndon, and J. F. Hughes. SKETCH: An interface for sketching 3D scenes. In *SIGGRAPH '96: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, pages 163–170. ACM Press, 1996.
144. J. Y. Zheng and S. Suezaki. A model based approach in extracting and generating human motion. In *Proceedings of 14th International Conference on Pattern Recognition*, volume 2, pages 1201–1205, Aug. 1998.
145. J. Y. Zheng and D. Takagi. Interactive human motion acquisition from video sequences. In *Proceedings of Computer Graphics International*, pages 209–217, 2000.

Index

Printed in the United States of America

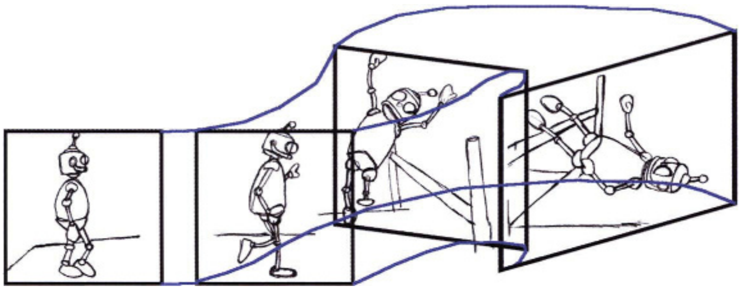


Fig. 1.4. The moving-camera frame.

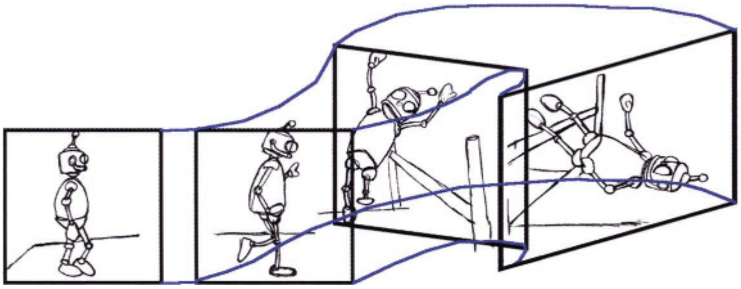


Fig. 1.5. Path of the camera center across all frames.

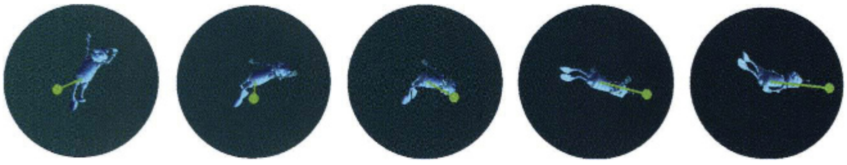


Fig. 2.8. The small sphere is the recovered camera position and the line shows the view direction vector. The larger sphere, centered at the look-at point, gives an idea of the relative positioning of the recovered camera centers.

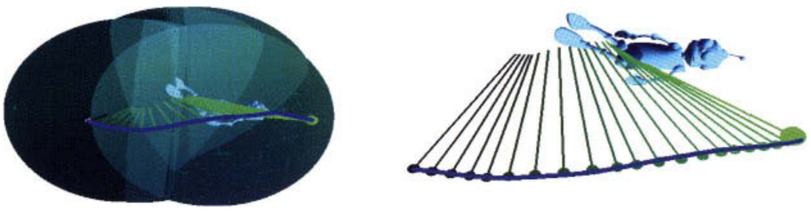


Fig. 2.9. The left image shows the path traced on the envelope of the view space. The right image shows a close-up view of the path. The larger green sphere at the end of the path shows the position of the (current) camera when this snapshot was captured.

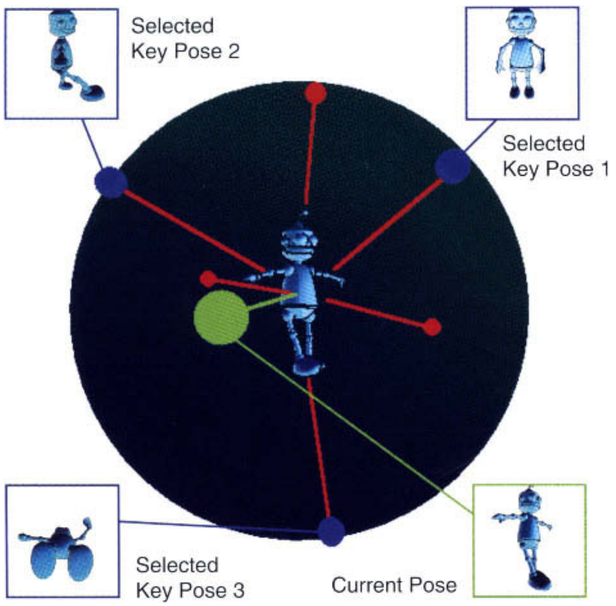


Fig. 3.13. The view space — the smaller blue and red spheres are the key viewpoints.

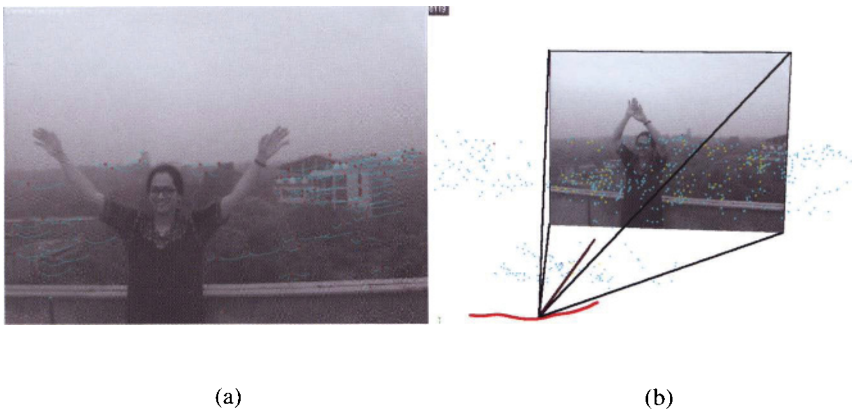


Fig. 4.5. (a) feature points tracked by Boujou, (b) camera tracking by Boujou. The camera path recovered is shown in *red*.

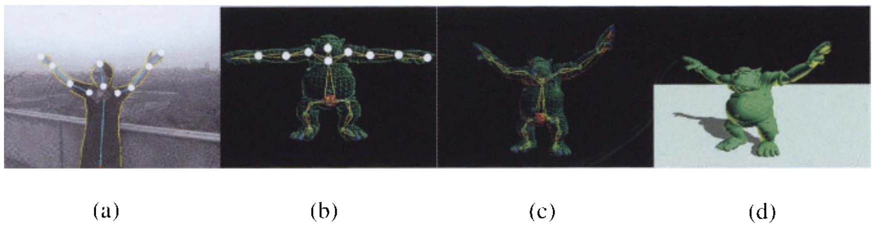
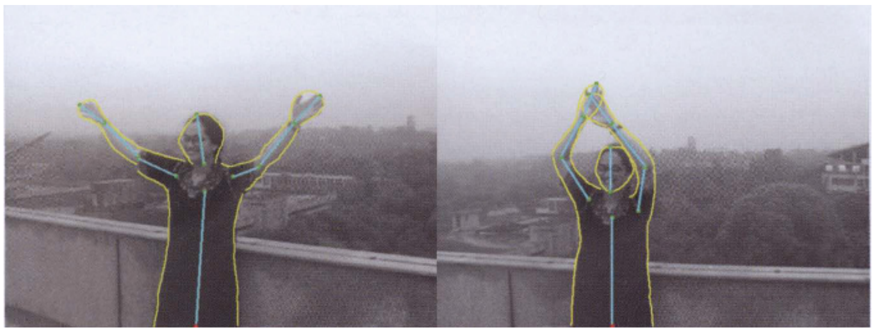
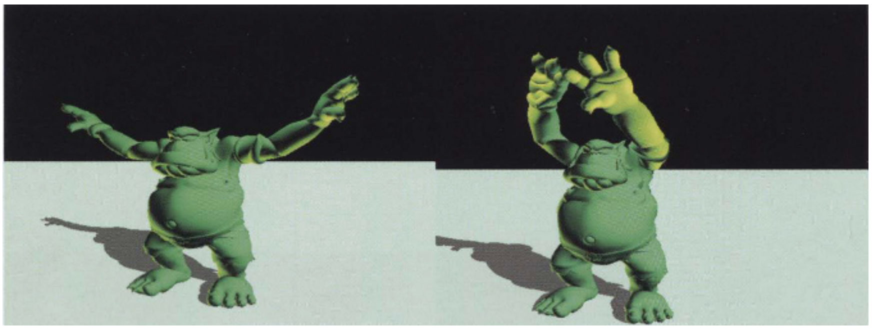


Fig. 4.6. Posing the character from a video frame: (a) contours tracked on a frame of the input video with joints of the 2D skeleton marked in *white*, (b) corresponding joints on the 3D skeleton marked in *white*, (c) 3D skeleton and character's mesh after posing, (d) final rendered pose of the character.



(a)



(b)

Fig. 4.7. (**a**) tracked contours and associated 2D skeletons on two key frames, (**b**) corresponding posed character viewed through their respective recovered cameras.

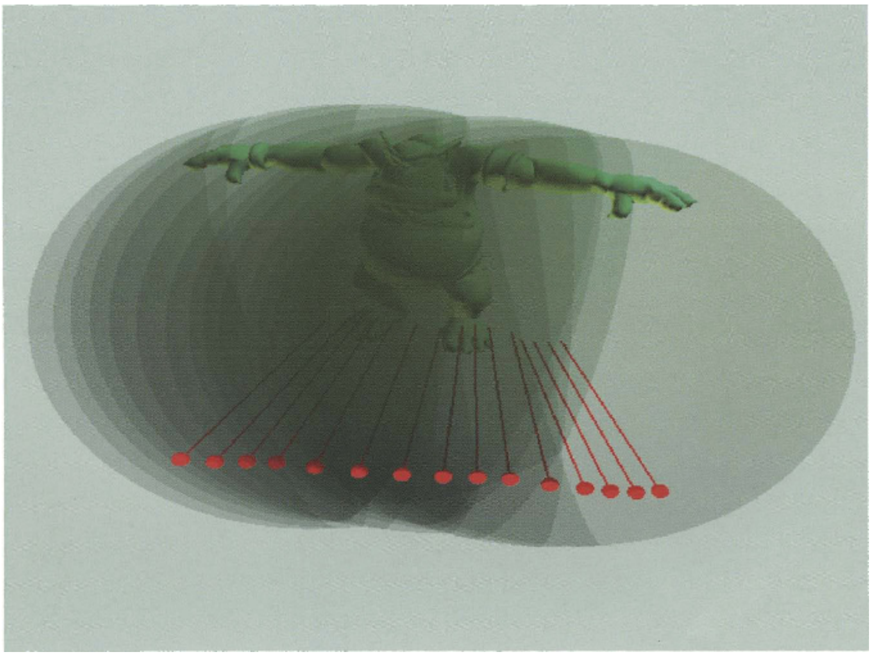


Fig. 4.8. The viewpoints, the view directions and the view space.

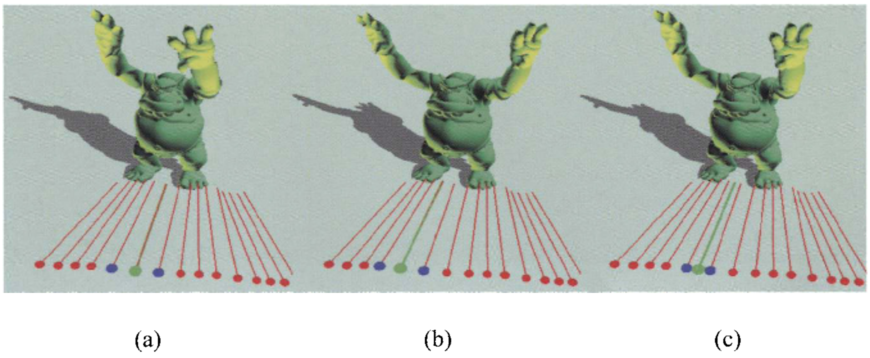


Fig. 4.9. Character poses associated with key views and novel view generation.

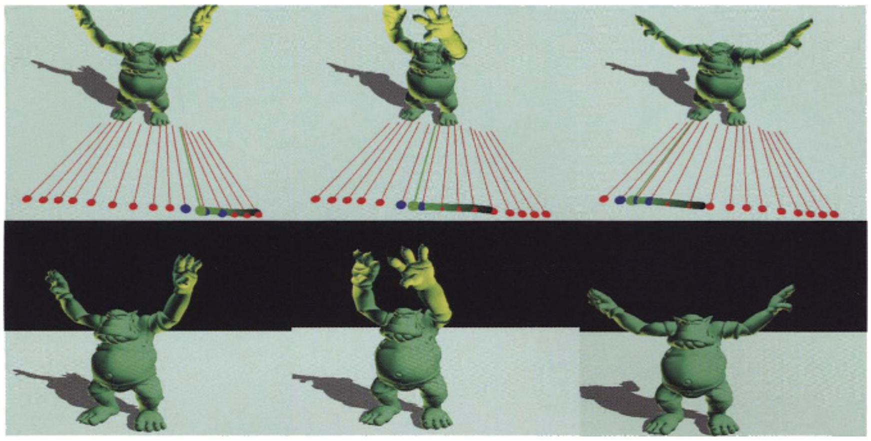
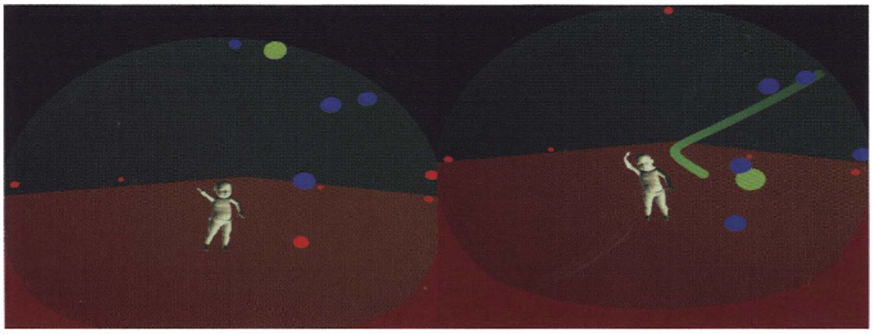


Fig. 4.10. The top row shows the camera path. The bottom row shows the corresponding generated animation frames.



(a) (b)

Fig. 4.13. (**a**) envelope of the view space constructed using the cameras recovered from the sketches, (**b**) transplanted camera path.

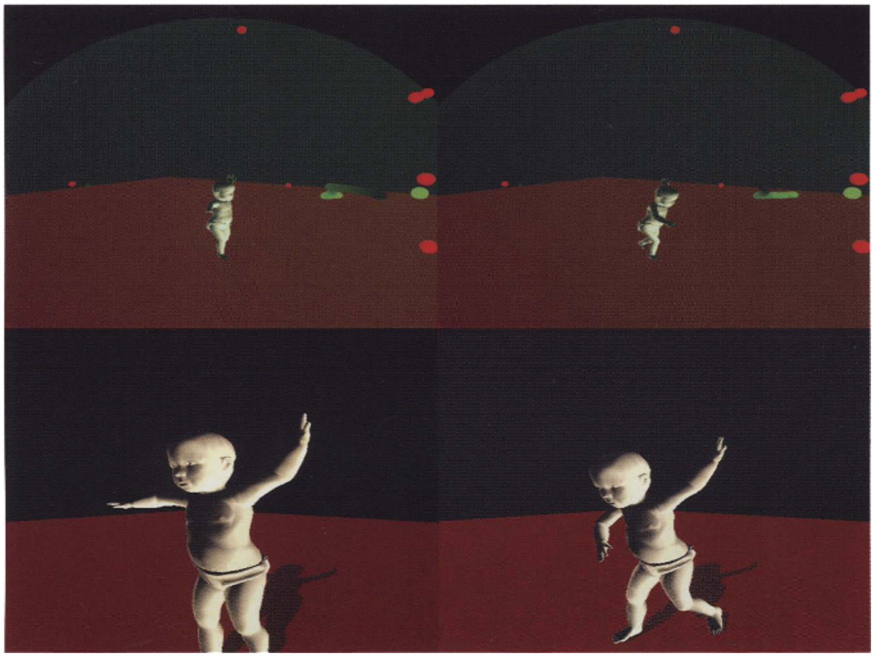


Fig. 4.16. The top row shows the camera path changing only in distance. The bottom row shows the corresponding generated animation frames.

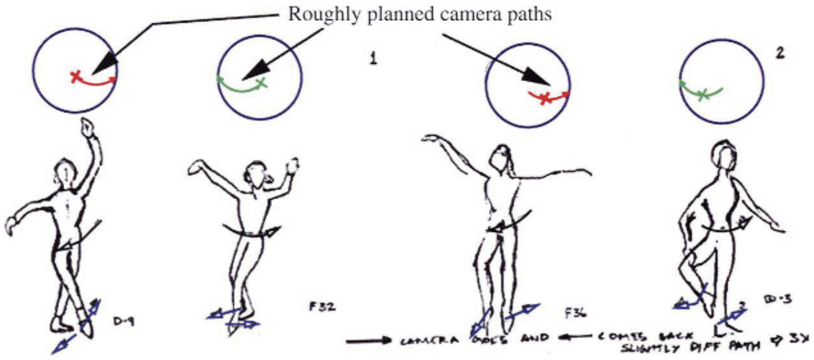


Fig. 5.4. Part of the storyboard for the reuse animation example.

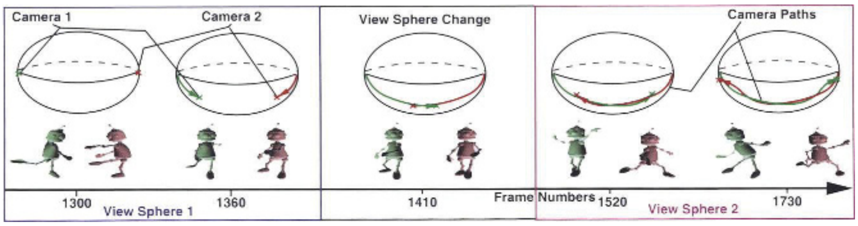


Fig. 5.7. Camera 1 (*in green*) generates the green character, Camera 2 (*in red*) generates the red character. Each view sphere generates two character poses in response to the two cameras.

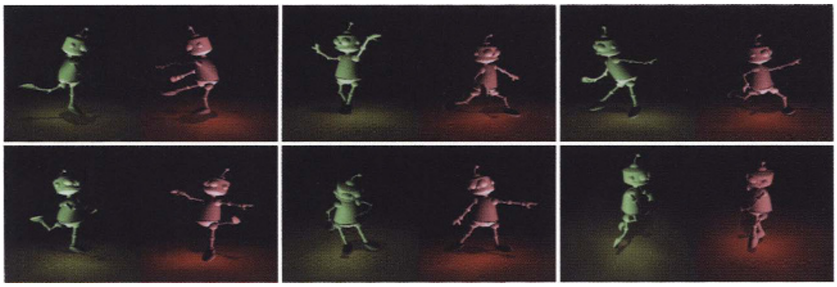
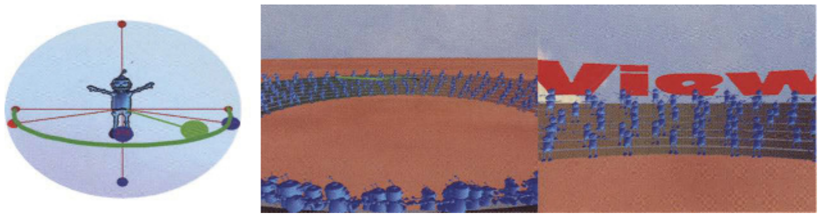


Fig. 5.8. Frames from the synthesized animation.



(a) (b) (c)

Fig. 5.10. The Mexican Wave Animation.