

Implementing Film Grammar With 3D Graphics

Aimin Zheng

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

April 2005

© Aimin Zheng, 2005



Library and
Archives Canada

Bibliothèque et
Archives Canada

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file *Votre référence*

ISBN: 0-494-10302-7

Our file *Notre référence*

ISBN: 0-494-10302-7

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.


Canada

Abstract

Implementing Film Grammar With 3D Graphics

Aimin Zheng

In film conversation shots, there are constraints that specify the players' positions as they are projected on the screen. How to project the two players on the appropriate position on the screen is the main problem focused on this thesis.

Some important rules of film grammar for two players are stated in this paper and an algorithm based on camera control to achieve desired projected position on screen for players are provided. The results of the algorithm for lots of shots in two players conversation satisfy all the constraints related with camera placement for two players in film grammar.

Translating film grammar into mathematic requirement for 3D graphics and building the corresponding equations are showed in this paper. And a creative and effective way to solve these equations is stated. This algorithm provides a feasible way to control the camera in graphics environment. And several cinematographic shots are implemented by this algorithm, and also a sequence of shots is applied into a film scene.

Acknowledgement

I want to express my gratitude to my supervisor Prof. Peter Grogono for his constant help during my thesis work. First, he introduced this wonderful research topic to me. Second, he provided me lots of helpful suggestions when I program for this thesis. Next, I would thank him for his creative ideas to resolve the problems during algorithm design. Last, for his structure advices and lots of revisions and comments to my thesis. He helped me a lot. And I would like to say: Thank you, Dr. Grogono.

I would also like to thank my family, for their patience and endless support during my study in this program at Concordia University. Without their supporting and encouraging me, I wouldn't finish this thesis.

Contents

List of Figures	
List of Tables	
Chapter 1 Introduction	1
1.1 Graphics and Film Grammar	1
1.2 Problem description	2
1.3 Contribution of thesis	2
Chapter 2 Related Work	4
Chapter 3 Background	6
3.1 Cinema grammar	6
3.1.1 Camera Placement	7
3.1.1.1 Line of Interest	7
3.1.1.2 The Triangle Principle	7
3.1.1.3 Importance of the Heads	9
3.1.1.4 Five basic variations of the triangle principle	9
3.1.1.4 Camera Placement	14
3.1.1.5 Camera Movement and Distance	15
3.1.2 Heuristics and constraints	15
3.1.3 Idioms Related With Dialogue Between Two Players	16
3.1.3.1 Both Players Sit Facing Each Other	16
3.1.3.2 One Player Sits While the Other Stands	18
3.1.3.3 Player A Moves Towards Player B	18
3.2 Computer Graphics – OpenGL	20
3.2.1 About OpenGL	20
3.2.2 Model View and Projection	22
3.2.2.1 Coordinate System and transformation	22
3.2.2.2 Model View Transformation	24
3.2.2.3 Projection Transformation	26
3.2.2.4 Viewport Transformation	28
3.2.2.5 OpenGL Function gluLookAt()	29
Chapter 4 Design	34
4.1 Camera module design	34
4.1.1 Camera components	34
4.1.2 Cinematic camera position algorithm design	37
4.2 Design of an Animated Conversation	47
4.2.1 Project Components	47
4.2.2 Film scenes to be animated	48
Chap 5 Implementation	49

5.1 User interface implementation	49
5.2 Camera algorithm Implementation	54
5.2.1 Data structure	54
5.2.2 Algorithms Implementation	57
5.2.2.1 Camera Movement.....	57
5.2.2.2 Collision detection	61
5.2.2.3 Get an ideal camera position according to user input parameters	62
5.3 Program implementation.....	63
5.3.1 Human being implementation	63
5.3.1.1 Sit down animation	66
5.3.1.2 Stand animation	66
5.3.1.3 Walk animation	66
5.3.2 Furniture (table and chair) implementation	66
5.3.3 Room and ornament picture implementation	67
5.4 Film scenes animation implementation	67
Chapter 6 Results.....	72
6.1 Film scenes animation results	72
6.1.1 Conversation when both people sit down	72
6.1.2 Conversation when one sits down, another stands up	74
6.1.3 Red moves toward Blue	76
6.1.3.1 Three-shot animation	76
6.1.3.2 Four-shot animation	78
6.2 Cinematic camera position algorithm results.....	79
Chapter 7 Conclusion and future works.....	84
7.1 Conclusions.....	84
7.2 Future work.....	86
References	87
Web based references.....	89

List of Figures

FIGURE 3.1 TWO TRIANGULAR FORMATIONS ON EACH SIDE OF THE LINE OF INTEREST	8
FIGURE 3.2 EXTERNAL REVERSE ANGLES.....	10
FIGURE 3.3 INTERNAL REVERSE ANGLES.....	11
FIGURE 3.4 SUBJECTIVE CAMERA ANGLES.....	11
FIGURE 3.5 PARALLEL POSITIONS.....	12
FIGURE 3.6 RIGHT ANGLE POSITIONS	12
FIGURE 3.7 THE RIGHT ANGLE CAMERA POSITIONS CAN ALSO BE BEHIND THE PLAYERS.....	13
FIGURE 3.8 ADVANCE ON A CAMERA COMMON VISUAL AXIS.....	13
FIGURE 3.9: CAMERA PLACEMENT IS SPECIFIED RELATIVE TO THE “LINE OF INTEREST”	14
FIGURE 3.10 FILMING A CONVERSATION BETWEEN TWO PLAYERS.....	17
FIGURE 3.11 A THREE-SHOT IDIOM OF ONE PLAYER APPROACHING ANOTHER.....	19
FIGURE 3.12 A FOUR-SHOT IDIOM OF ONE PLAYER APPROACHING ANOTHER	20
FIGURE 3.13 DEMONSTRATION OF MODELING COORDINATES BEING TRANSFORMED TO	23
DEVICE COORDINATES FOR A THREE-DIMENSIONAL SCENE	23
FIGURE 4.1 CAMERA COMPONENTS	35
FIGURE 4.2 BOUNDING BOX FOR OBJECTS INSIDE A ROOM	36
FIGURE 4.3 BOUNDING BOX FOR A ROOM	37
FIGURE 4.4 TRIANGLE FORMED BY TWO PLAYERS AND THE CAMERA	39
FIGURE 4.5 PERSPECTIVE PROJECTION FRUSTUM OF TWO PLAYERS AND CAMERA	40
FIGURE 4.6 PROGRAM COMPONENTS	47
FIGURE 5.1 PROGRAM INTERFACE	50
FIGURE 5.2 CROSS PRODUCT OF UP AND MODEL - EYE OR EYE - MODEL.....	57
FIGURE 5.3 CROSS PRODUCT OF UP AND MODEL-EYE	59
FIGURE 5.4 CAMERA ROTATE AROUND UP VECTOR.....	60
FIGURE 5.5 ANIMATED ROOM SCENE.....	63
FIGURE 5.6 A SHOT - CAMERA LOOKS AT RED.....	68
FIGURE 5.7 A SHOT - CAMERA LOOKS AT BLUE.....	69
FIGURE 5.8 SHOT 1 OF THREE-SHOTS IDIOM.....	70
FIGURE 5.9 TWO FILM SHOTS OF FOUR-SHOTS IDIOM. SHOT 1 AND SHOT 3.....	71
FIGURE 5.10 TWO SHOTS FOR TWO PLAYERS. ONE IS STANDING WHILE ANOTHER IS SITTING	71
FIGURE 6.1 SCENE 1 SHOT 1 – FACING RED	73
Figure 6.2 Scene 1 shot 2 – facing Red	73
FIGURE 6.3 SCENE 1 SHOT 3– LOOK AT BLUE	74
Figure 6.4 Scene 1 shot 4 – look at Blue.....	75
FIGURE 6.5 SCENE 2 SHOT 1, FACING RED	74
Figure 6.6Scene 2 shot 2 facing Red.....	75
FIGURE 6.7 SCENE 2 SHOT 3,FACING BLUE	75

Figure 6.8 Scene 2 shot 4, facing Blue.....	76
FIGURE 6.9 SCENE 2 SHOT 5, LOOK AT RED	75
Figure 6.10 Scene 2 shot 6, look at.....	76
FIGURE 6.11 SCENE 3 SHOT 1. LOOK AT RED, AND RED GOES AWAY FROM SCREEN	77
FIGURE 6.12 SCENE 3 SHOT 2. RED ENTERS INTO SCREEN FROM RIGHT	77
FIGURE 6.13 SCENE 3 SHOT 3. CAMERA MOVES FORWARD CLOSER TO BOTH.....	78
FIGURE 6.14 SCENE 4, SHOT 1	79
Figure 6.15 Scene 4, shot 2.....	80
FIGURE 6.16 SCENE 4 SHOT 3	79
Figure 6.17 Scene 4 shot 4.....	80
FIG. 6.18 SHOT 1. $k = 1.28$, ANGLE = 90	81
Fig. 6.19 Shot 2 $k=2.7$, angle = 270.....	82
FIG. 6.20 SHOT 3. $k=1.7$, ANGLE = 264	81
Fig. 6.21 Shot 4. $k=2.3$, angle = 60.....	82
FIG. 6.22 SHOT 5. $k=2.1$, ANGLE = 81	81
Fig. 6.23 Shot 6. $k=2.1$, angle = 81.....	82
FIG. 6.24 SHOT 7. $k=1.0$, ANGLE = 210	82
Fig. 6.25 Shot 8. $k = 0.7$, angle = 266.....	83
FIG. 6.26 SHOT 9. $k = 0.4$, ANGLE = 196	82
Fig. 6.27 Shot 10. $k = 1.75$, angle = 275.....	83
FIG. 6.28 SHOT 11. $k = 1.05$, ANGLE = 84.5	82
Fig. 6.29 Shot 12. $k=3.5$ angle = 90.....	83
Fig. 6.30 Shot 13. $k=1.0$ angle = 90.....	84

List of Tables

TABLE 1. PARAMETERS FOR CAMERA POSITION CALCULATING IN SCENE 1.....	73
TABLE 2. PARAMETERS FOR CAMERA POSITION CALCULATING IN SCENE 2.....	76
TABLE 3. PARAMETERS FOR CAMERA POSITION CALCULATING IN SCENE 3.....	77
TABLE 4. PARAMETERS FOR CAMERA POSITION CALCULATING IN SCENE 3.....	78
TABLE 5. PARAMETERS FOR CAMERA POSITION CALCULATING SOME SEPARATE SHOTS.....	80

Chapter 1 Introduction

1.1 Graphics and Film Grammar

Computer graphics is the field of visual computing, which utilizes computers both to generate visual images synthetically and to integrate or alter visual and spatial information sampled from the real world. Advances in computer technology have led to widespread use of computer graphics, especially in the entertainment industry. Within the computer graphics field, camera control and movements have long been studied in many different contexts; for animation; for exploration/manipulation, or for presentation.

Film is a form of art expression used primarily for story telling. “Film language was born when film makers became aware of the difference between the loose joining together of small images in various state of motion, and the idea that these series of images could be related to one another” [Arijon 76]. Film directors have implicitly established a set of rules and conventions that encode proper ways to tell stories in an accurate and explicit mode.

In the trend of current computer animation and games, camera manipulation become more and more important since program designers wish both players and viewers to be able to feel the atmosphere deeply while playing the role or watching the presentation. Good camera techniques can enhance the viewer’s experience. A cinema grammar is a set of rules and conventions to tell story clearly. Since the film industry has designed cinema

grammar to direct camera control in film-making, how to apply film language into camera control in computer graphics has attracted much attention in computer graphics research in recent years. The research described in this thesis shows that controlling the camera in a cinematographic way can always show the viewers the most salient scenes in an interactive graphics environment without bothering the viewers with the need to issue camera control commands.

1.2 Problem description

In film conversation shots, there are constraints that specify the players' positions as they are projected on the screen. How to project the two players on the appropriate position on the screen is the main problem we will focus on in this thesis. In order to get the desired visual effect, we must place the camera position according to the rules of film grammar. In this thesis, we provide algorithms for computing appropriate camera positions. The results of each algorithm must satisfy all the constraints related with camera placement in film grammar. Film grammar rules for two players will be introduced in this thesis, and we will also implement some film scenes based on camera control provided by the algorithm.

1.3 Contribution of thesis

This thesis describes the construction of a camera framework, and concentrates on methods for satisfying multiple constraints on the camera position in filming two players conversation. This camera framework is applied to obtain some visual effects satisfying film grammar in several film scenes.

In this thesis, some important rules of film grammar are stated and algorithms are provided to achieve the desired projected positions on the screen for players. We show how to translate the film grammar into mathematical requirements for 3D graphics and how to derive the corresponding equations. We also find a creative and effective way to solve these equations. We show that the algorithms provide a feasible way to control the camera in graphics environment and complete several cinematographic shots, and to apply a sequence of shots into a film scene. The algorithms we provide in this thesis is useful for game program developer or film makers to do story boarding.

Chapter 2 Related Work

The subject of using film grammar to control camera positions and scene structure has received relatively little attention in computer graphics and AI communities. Some related works are listed below.

Drucker [Drucker 95] suggested a method of encapsulating camera tasks into well defined units called “camera modules”. In this paper, the problem of setting up the optimal camera position for individual shots and subject to constraints was suggested. A camera module which can film a conversation between two virtual actors was described. Some camera placement constraints related with two players conversation film were also listed. But no algorithm for camera placement was suggested.

Christianson *et al.* [Christianson 98] described several principles of cinematography and showed how they can be formalized into a declarative language, called the *Declarative Camera Control Language* (DCCL). And they applied the DCCL within the context of a simple interactive video game. By encoding 16 idioms from a film textbook, he argued that DCCL represents cinematic knowledge at the same level of abstraction as expert film directors. These idioms produce compelling animations. A set of possible cameras is fully specified by the shot descriptions in DCCL and the geometry of the scene. The final selection from among this set of different shots is made according to how well each shot covers the scene.

Lin *et al.* [Lin 04] proposed a mechanism of camera control in 3D computer games.

The system can automatically direct the camera based on some cinematic heuristics. Because a game can usually be decomposed into several specific scenes which often occur in motion pictures, they used a sequence of shots similar to cinematic heuristics to describe the camera behavior in a scene. The control mechanism collects and analyses information from the game content and automatically directs the camera to capture event scenes. They encapsulated cinematic camera techniques into camera modules and used a finite-state machine model to encode the procedure of shooting a scene into a description of shots. Therefore, the system can also assist designers to add effects of cinematic camera control by providing camera modules and descriptions of shots. The concept of frame-coherence is integrated into their system for smooth camera movement . With all these features, this camera control module can automatically generate shots and arrange these shots to provide a cinematic effect suitable for game playing.

Bares *et al.* [Bares 97] developed UCAM (User-Customized Automated Montage), a real-time camera planner that employs cinematographic user models to render customized visualizations of dynamic 3D environments. Bares developed a user-sensitive realtime camera planner, User-Customized Automated Montage (UCAM). UCAM creates customized camera control to plan camera positions, view directions, and camera movement through its cinematographic user model. In the meantime, users can adjust the camera control by their visualization preference based on this cinematographic user model.

Chapter 3 Background

In this chapter, we will introduce background of cinema grammar and OpenGL graphics principle.

Arijon [Arijon76] provides straightforward descriptions for filming any of a large number of situations. These form a perfect basis for deriving a large class of camera primitives. This class of primitives consists of projection constraints that involve the placement of the camera based on the projection of an object onto the screen.

OpenGL graphics produces a series of coordinate transformations: model view transformation, projection transformation, and viewport transformation. These transformations are combined together to make an OpenGL scene on the screen. And OpenGL provides function `gluLookAt()` for users to control the camera. Arijon provides us cinematic grammar to control camera in filming, while OpenGL provides us a function to control camera in computer graphics, so we can apply these cinematic camera rules to computer graphics camera control.

3.1 Cinema grammar

A film can be considered to be a sequence of frames, and it is often helpful to think of a film as having structure. At the highest level, a film is a sequence of *scenes*, each of which captures some specific situation or action. And each scene in the film is composed of one or more *shots*. A single shot covers the small portion of a movie between when a

camera is turned on and when it is turned off. Typically, a film is comprised by a large number of individual shots with each shot's duration lasting from a second or two in length to perhaps tens of seconds [Christianson 96].

3.1.1 Camera Placement

3.1.1.1 Line of Interest

All dialogue scenes have two central players. The *line of interest* is an imaginary vector connecting two interacting players. It is directed along the line of an actor's motion, or oriented in the direction that the actor is facing. The *line of interest* between two central players in a scene is based on the direction of the looks exchanged between them [Arijon 76].

3.1.1.2 The Triangle Principle

We can observe a line of interest from **three extreme positions** without crossing to the other side of the line (see triangle in Figure 3.1, note the symbol represents a human figure-the flat side indicates the face of the figure). These **three extreme positions** form a triangular figure with its base parallel to the line of interest [Arijon 76]. The importance of this triangle is that each performer (X or Y) is framed on the same side of the screen in each shot with player X on the left side and player Y on the right.

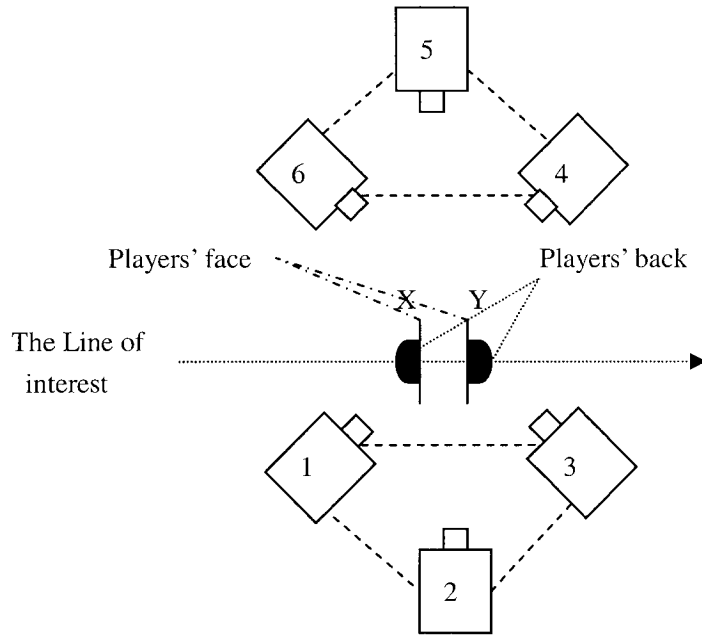


Figure 3.1 Two triangular formations on each side of the line of interest.

Two triangular formations can be employed one on each side of the line of interest. One of them has to be chosen, excluding the position on the other.

From Figure 3.1, we know that two triangular camera formations can be set, one on the other side of a line of interest which is symmetric to the triangle in Figure 3.1 along the line of interest.

But we cannot successfully cut from a camera position in one pattern to another on the other triangular arrangement. If we do that, we will only confuse our audience, because using two camera positions located on different triangular formations will not present a steady emplacement of the players on the same areas of the screen.

According to Arijon, a **cardinal rule** for the *triangular camera principle* is to select one side of the line of interest and stick to it. This is one of the most respected rules in film language.

3.1.1.3 Importance of the Heads

It is usually quite simple to draw the line of interest flowing between two players when they are standing face to face, or sitting facing each other. But when the actors are lying down with their bodies parallel or extended in opposite directions, it seems more difficult. However, if we remember only that the central points of two persons talking to each other are their heads, then it is quite simple.

In film grammar, the positions of the bodies do not really count, it is the heads that matter. They attract our attention immediately, regardless of the positions of the bodies, because the head is the source of human speech and the eyes are the most powerful direction pointers that a human being has to attract or direct interest. Even when one actor has his back to the other, or they are back to back, a line of interest passes between their heads. In all film scenes, the line of interest must flow along the line between the heads of the two central performers.

3.1.1.4 Five basic variations of the triangle principle

There are five variants of the triangle principle: 1) external reverse angles, 2) internal reverse angles, 3) parallel positions, 4) right angle positions and 5) common visual axis. I will examine each of them separately.

The two sites on the base of the triangular camera locations (parallel to the line of interest of the scene), provide the three variations with which a linear disposition of the players can be covered. The cameras placed on those two viewpoints can be pivoted on

their axis, obtaining three well differentiated positions. Each one of those positions is applied in pairs. Both camera angles on the base of the geometric figure assume identical positioning in their relation to the players covered.

External Reverse Angles.

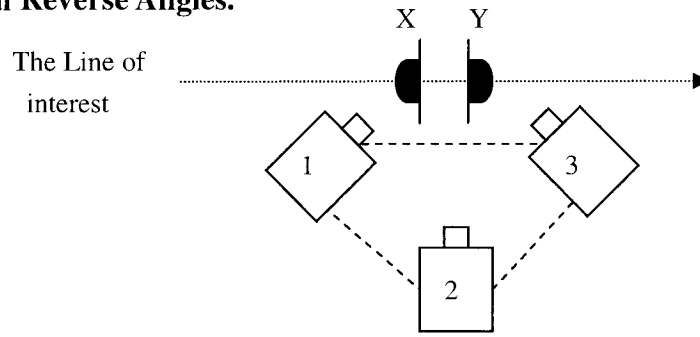


Figure 3.2 External reverse angles.

The cameras in the two positions parallel to the line of interest are directed inward towards the players.

In this first variant, both camera positions on the base of the triangle are behind the backs of the two central players, angled in, close to the line of interest between the performers and covering them both (Fig. 3.2).

Internal Reverse Angles. In the second variant, the cameras are between the two players, pivoted outwards from the triangular figure, and close to the line of interest though not representing the viewpoints of the performers (Fig. 3.3). In either case the rapport is not that of a head-on confrontation, though quite close to it in effect.

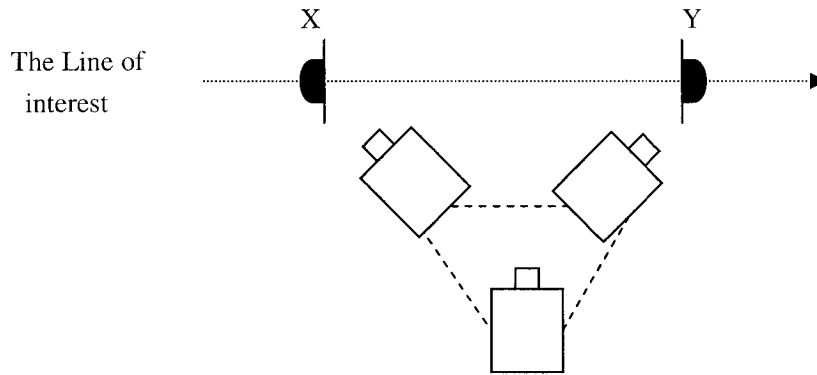


Figure 3.3 Internal reverse angles.

In this variant the two camera positions parallel to the line of interest point outwards, covering each player individually.

Fig. 3.4 is a form of internal reverse with the cameras back to back to represent the subjective viewpoint of the player excluded from the shot.

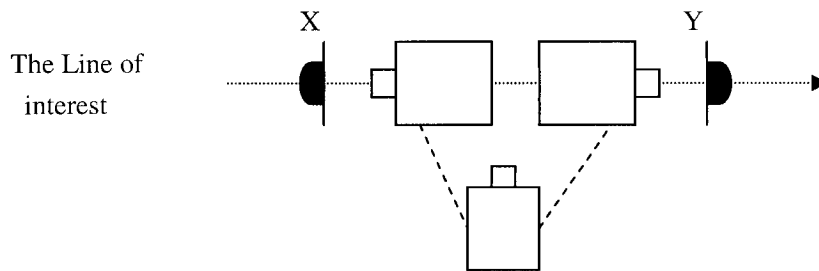


Figure 3.4 Subjective camera angles.

If the camera positions are back to back on the line of interest itself, they each become the subjective point of view of the player excluded from the shot.

Parallel Positions. In the third variant the camera sites are on the base of the triangular figure close to the line of interest, deployed with their visual axes in parallel, and covers the players individually.

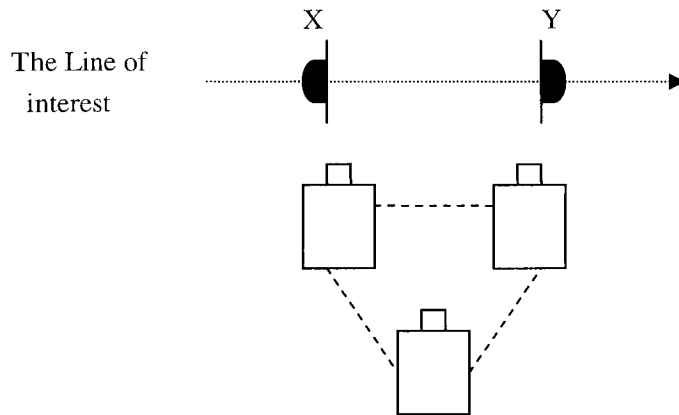


Figure 3.5 Parallel positions.

When both camera positions have their visual axis in parallel, they cover each player individually giving us a profile view.

Right Angle Positions. When the actors are placed side by side in an ‘L’ formation, the camera viewpoints on the base of the imaginary triangle acquire a right angle relationship, close to the line of interest passing between the players. In this case, the camera is in front of the players (Fig. 3.6).

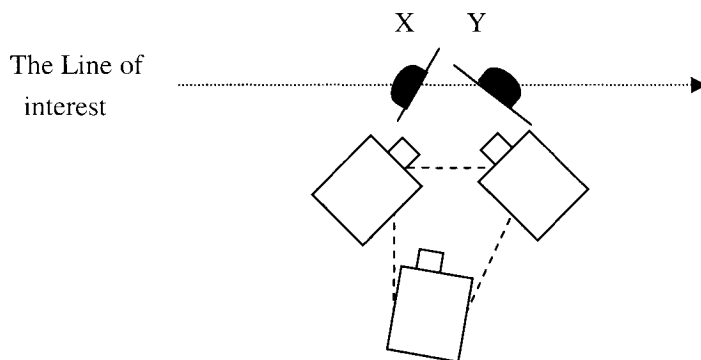


Figure 3.6 Right angle positions

When the players are placed side by side in an L formation, a right angle camera relationship is assumed by the two sites located on the base of the triangular figure for camera placement.

The same arrangement can be placed behind the players, with which a new variant

for dialogue coverage is achieved, shown in Fig. 3.7.

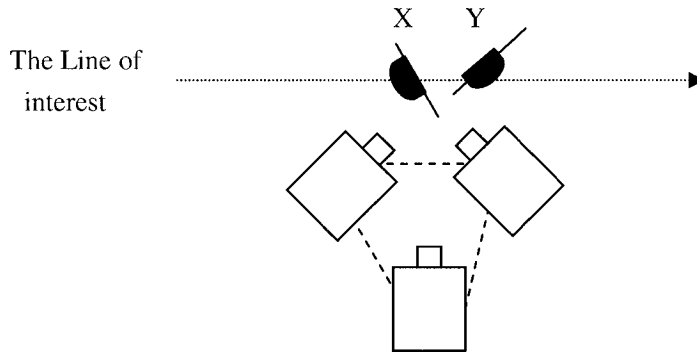


Figure 3.7 The right angle camera positions can also be behind the players

Common Visual Axis. To cover only one of the players in a shot while framing both players on the other, the camera in one of the two viewpoints on the triangle base, must be advanced on its visual axis.

Advancing on either of the two viewpoints (optically or physically) we obtain a closer shot of the selected player, thus emphasizing him/her over his/her partner. Fig. 3.8 shows the arrangement.

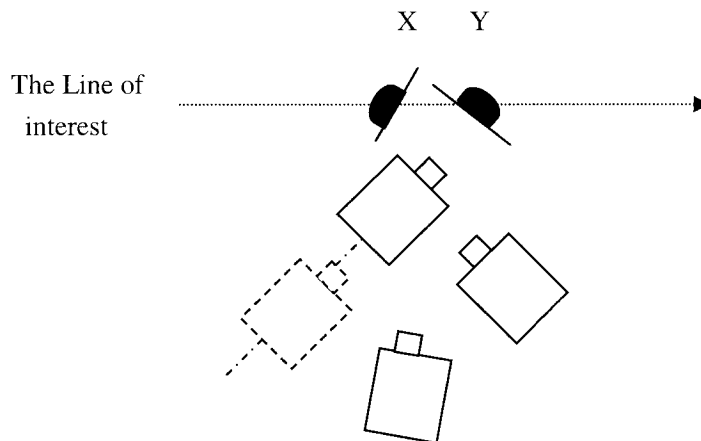


Figure 3.8 Advance on a camera common visual axis.

To obtain coverage of a single player in the group, one of the cameras is moved forward on the visual axis line of either of the two positions on the base of the triangle.

The above mentioned five basic variations are used not only to cover static conversations of a group of players, but also the movement of those players on the screen.

3.1.1.4 Camera Placement

Film directors specify camera placements relative to the *line of interest*. Figure 3.9 shows the first three triangle variants can be combined to multiply the camera placement.

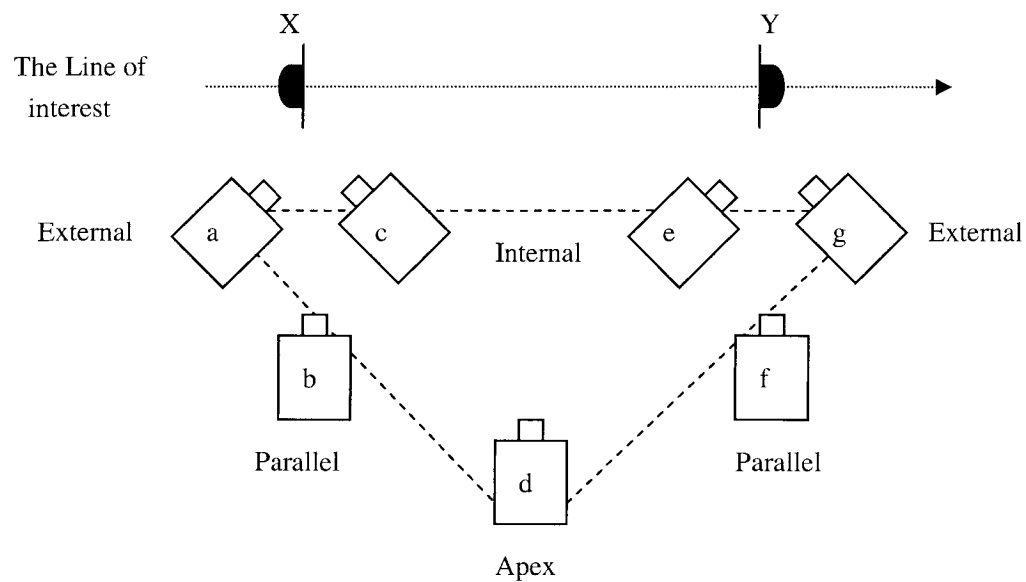


Figure 3.9: Camera placement is specified relative to the “line of interest”
(Adapted from figure 4.3 and figure 4.11 of Arijon 1976)

Shooting actor X from camera position **b** and **f** gives a *parallel camera* placement. Filming X from position **c** and **e** yields an *internal reverse* placement. Shooting from position **d** results in an *apex* shot that shows both actors. Finally, filming from **a** or **g** gives an *external reverse* placement. All positions can be combined in pairs to cover both players, except for the internal and parallel sites that cover each of the subjects individually.

3.1.1.5 Camera Movement and Distance

During a shot the camera can remain fixed, or it can pan (rotate about a vertical axis, so the image appears to move horizontally), or it can tilt (rotate about a horizontal axis, so the image appears to move vertically), or it can track (travel at different speeds attached to a moving vehicle). It moves to support the action that it records.

The camera can do all the above movements from different distances. Cinematographers have identified that certain “cutting heights” make for pleasing compositions while others yield ugly results (e.g., an image of a man cut off at the ankles). Roughly there are five basic definable camera distances. An extreme close up shows the head and cuts at the neck; a close up cuts under the chest or at the waist; a medium view cuts at the crotch or under the knees; a full view shows the entire person; and a long view provides a distant perspective [Arijon 76].

3.1.2 Heuristics and constraints

In this thesis, I surveyed and picked some important principles as considerable factors of *camera control*. And they are general rules in film grammar, not only apply to two people, but also more than two. These principles are listed in [Lin 04] [Christianson 96]:

Parallel editing: Visualized scenes in a film should alternate between different characters, locations, or times.

Only show peak moments of the story: Repetitive moments from a narrative should be deleted.

Triangle principle (Do not cross the line): Once an initial shot is taken from the left or right side of the line of interest, subsequent shots should remain on the same side. This rule ensures that the direction of the actor's motion is clear. Again this is the triangle principle.

Let the actor lead: The actor initiates all movement, and then the camera follows. Therefore, camera setup should be considered before the actor.

Break movement: A scene illustrating motion should be broken into two shots. When the actor appears to move across the middle of the screen, a shot is often cut to another shot.

3.1.3 Idioms Related With Dialogue Between Two Players

Perhaps the most significant invention of cinematographers is the notion of *cinematic idiom* – a stereotypical way to capture some specific action as a series of shots.

3.1.3.1 Both Players Sit Facing Each Other

For example, in a dialogue between two people, a filmmaker might begin with an apex view of both actors, and then alternate views of each, at times using internal reverse placements and at times using external reverse.

When two players face each other, the strongest camera positions to record their dialogue are located on the base of triangle, parallel to the line of interest. Position **a** and **g** of the *external reverse* camera arrangement, have two immediate advantages over the

camera site situated on the *apex* of the triangle (position **d**). The first advantage is that they give composition in depth, because from their viewpoints, the actors are placed on two different planes: one close to the camera and the other further back.

The second advantage is that one of the actors faces the camera, getting the viewer's full attention, while the other has his/her back to us. In theatrical terms, the second actor has an open body position (face to the audience), while the first has a closed body position (his/her back to the audience). Therefore the player facing the camera is the dominant one.

On the screen this is accentuated further by the distribution of screen space in the composition of the shot, as shown in figure 3.10.

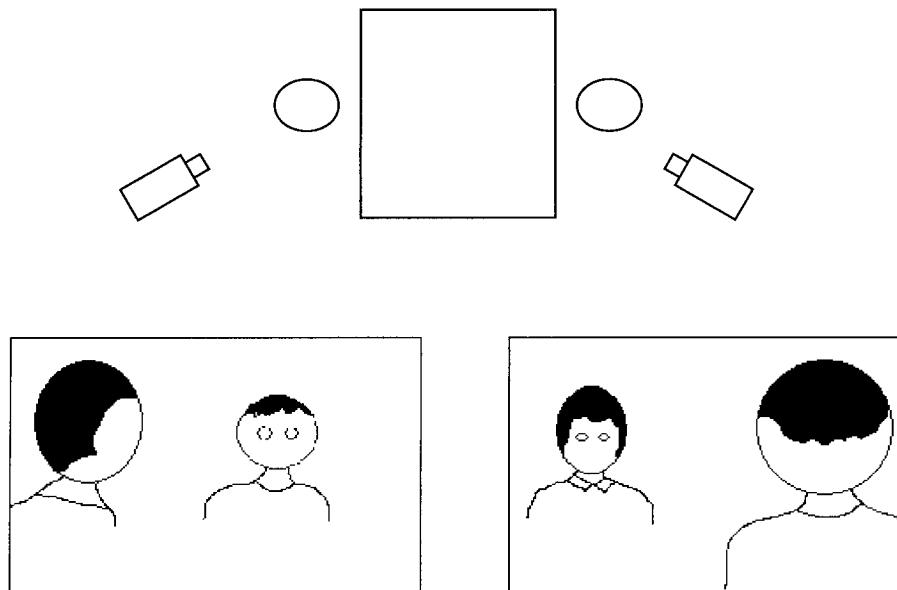


Figure 3.10 Filming a conversation between two players

On normal screen sizes, the actor who speaks is given two-thirds of the screen space, while his/her partner has only one-third.

While there is an infinite variety of idioms, film directors have learned to rely on a small subset of these. Indeed, film grammar books (Arijon 1976) are primarily a compilation of idioms along with a discussion of the situations when a filmmaker should prefer one idiom over another.

3.1.3.2 One Player Sits While the Other Stands

Camera height influences presentation. In the previous situation that two players sit face each other , the lens is usually at the same height as the actors' heads .

If one actor stands and the other is sitting, the camera height can vary for the reverse shot.

In the previous examples when two players sit facing each other, we use *external reverse* angles. When one of them sits while another stands, we can also use external reverse camera placement. If internal reverse positions are used to cover the same situation (one actor stands, the other sits) for single shots of each player the camera is alternately high and low, as if seeing the scene from each player's viewpoint.

3.1.3.3 Player X Moves Towards Player Y

The number of visual permutations possible for one player approaching another or a group is almost limitless. I will describe two approaches for basic situations that are used very often in this situation.

Three-shot Idiom

Figure 3.11 presents a three-shot idiom. The idiom, adapted from Arijon (1976), provides a method for depicting short-range motion of one actor approaching another. The first shot is a *close up*; actor X begins in the center of the screen and exits left. The second shot begins with a long view of actor Y; actor X enters from off-screen right, facing Y. Y may be either standing or sitting. The final shot begins with a medium view of Y, with actor X entering from off-screen right and stopping close to Y.

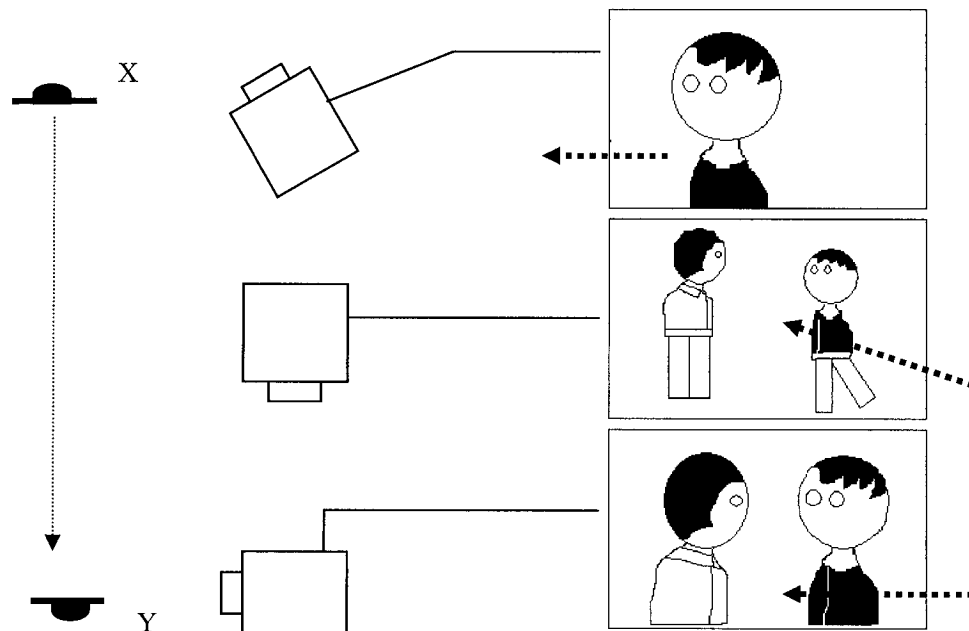


Figure 3.11 A three-shot idiom of one player approaching another

Four-shot Idiom

Figure 3.12 shows a four-shot idiom to depict short range motion of one actor approaching another. The idiom adapted from Arijon (1976), make the first shot to be actor X walks straight towards the camera site. Then the camera pans to look at his side

in the second shot. In the last two shots, the moving actor X begins his walk with his back to the camera and concludes by arriving at a profile position.

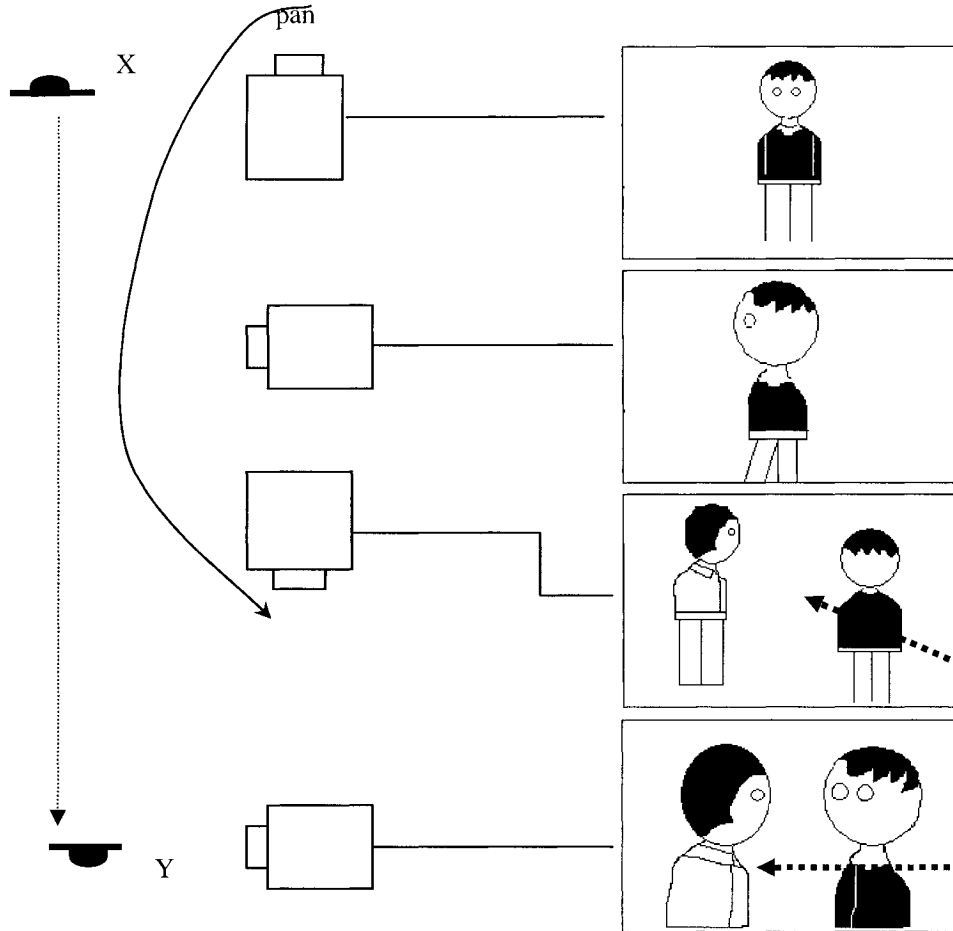


Figure 3.12 A four-shot idiom of one player approaching another

3.2 Computer Graphics – OpenGL

3.2.1 About OpenGL

OpenGL (Open Graphics Library) is a software interface to graphics hardware. It consists of three libraries: the Graphics Library (GL), the Graphics Library Utilities (GLU) and

the Graphics Library Utilities Toolkit (GLUT).

OpenGL is the premier environment for developing portable, interactive 2D and 3D graphics applications. Since its introduction in 1992, OpenGL has become the industry's most widely used and supported 2D and 3D graphics application programming interface (API), bringing thousands of applications to a wide variety of computer platforms. OpenGL runs on every major operating system including Mac OS, OS/2, UNIX, Windows 95/98, Windows 2000, Windows NT, Linux, OPENStep, and BeOS; it also works with every major windowing system, including Win32, MacOS, Presentation Manager, and X-Window System. OpenGL is callable from many programming languages, including Ada, C, C++, Fortran, Python, Perl and Java. It also provides complete independence from network protocols and topologies.

OpenGL doesn't provide high-level commands that describe models of three-dimensional objects. Such commands might allow you to specify relatively complicated shapes such as automobiles, parts of the body, airplanes, or ships. Using OpenGL, we can build our own models consisting of a series of geometric primitives such as point, line, or filled polygon, plus some modeling features provided by OpenGL Utility Library (GLU) such as quadric surfaces, NURBS curves and surfaces, and incorporate a broad set of rendering, texture mapping, special effects, and other powerful visualization functions. Developers can leverage the power of OpenGL across all popular desktop and workstation platforms, ensuring wide application deployment. All OpenGL applications produce consistent visual display results on any OpenGL API-compliant

hardware, regardless of operating system or windowing system.

3.2.2 Model View and Projection

We will introduce OpenGL transformation here. We will discuss the coordinate system, modelview transformation and projection transformation which are very important in OpenGL.

3.2.2.1 Coordinate System and transformation

OpenGL uses both the two-dimensional (2D) coordinate system and three-dimensional (3D) coordinate system but it is primarily intended for 3D applications.

In general, several different Cartesian reference frames are used in the process of constructing and displaying a scene. First, we can define the shapes of individual objects, such as trees or furniture, within a separate coordinate reference frame for each object. These reference frames are called *modeling coordinates*. Once the individual object shapes have been specified, we can construct a scene by placing the objects into appropriate locations within a scene reference frame called *world coordinates*. This step involves the transformation of the individual modeling coordinate frames to specified positions and orientations within the world-coordinate frame.

After all parts of a scene have been specified, the overall world-coordinate description is processed through various routines onto one or more output-device reference frames for display. This process is called the *viewing pipeline*. World-coordinate positions are first converted to *viewing coordinates* corresponding to

the view we want of a scene, based on the position and orientation of a hypothetical camera. Then object locations are transformed to a two-dimensional projection of the scene, which corresponds to what we will see on the output device. The scene is then stored in *normalized coordinates*, where each coordinate value is in the range from -1 to 1 . Normalized coordinates are also referred to as *normalized device coordinates*, since using this representation makes a graphics package independent of the coordinate range for any specific output device. We also need to identify visible surfaces and eliminate picture parts outside the bounds of the view we want to show on the display device. Finally, the picture is scan converted into the refresh buffer of a raster system for display. The coordinate systems for display devices are generally called *device coordinates*, or *screen coordinates* in the case of a video monitor. Often, both normalized coordinates and screen coordinates are specified in a left-handed coordinate reference frame so that increasing positive distance from the xy plane (the screen, or viewing plane) can be interpreted as being farther from the viewing position [Hearn 03].

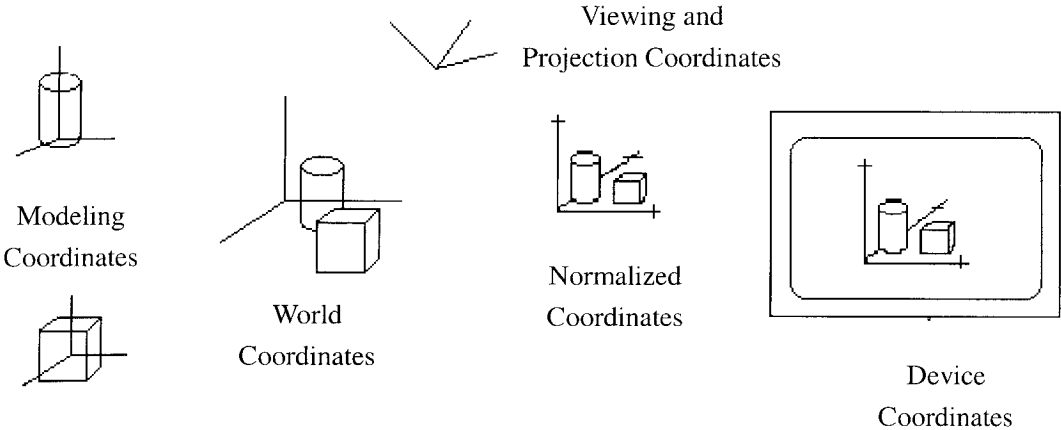


Figure 3.13 Demonstration of modeling coordinates being transformed to device coordinates for a three-dimensional scene

From the above description, we know that a series of three computer operations convert an object's three-dimensional coordinates to pixel positions on the screen:

1. Transformations, which are represented by matrix multiplication, include modeling, viewing, and projection operations. Such operations include rotation, translation, scaling, reflecting, orthographic projection, and perspective projection. Generally, we use a combination of several transformations to draw a scene.
2. Since the scene is rendered on a rectangular window, objects (or parts of objects) that lie outside the window must be clipped. In three-dimensional computer graphics, clipping occurs by throwing out objects on one side of a clipping plane.
3. Finally, a correspondence must be established between the transformed coordinates and screen pixels. This is known as a viewport transformation.

3.2.2.2 Model View Transformation

The viewing transformation is analogous to positioning and aiming a camera. The modeling transformation is to position and orient the model. In OpenGL, the viewing and modeling transformations are combined to form the modelview matrix, which is applied to the incoming object coordinates to yield eye coordinates.

The three OpenGL routines for *modeling transformations* are `glTranslate*`(),

`glRotate*()`, and `glScale*()`. These routines transform an object (or coordinate system) by moving, rotating, stretching, or shrinking it. All three commands are equivalent to producing an appropriate translation, rotation, or scaling matrix, and then calling `glMultMatrix*()` with that matrix as the argument. However, these three routines might be faster than using `glMultMatrix*()`. OpenGL automatically computes the matrices for us.

A *viewing transformation* changes the position and orientation of the viewpoint. Using the camera analogy, the viewing transformation positions the camera tripod, pointing the camera toward the model. Just like we move the camera to some position and rotate it until it points in the desired direction, viewing transformations are generally composed of translations and rotations. In order to achieve a certain scene composition in the final image or photograph, we can either move the camera, or move all the objects in the opposite direction. Thus, a modeling transformation that rotates the model counterclockwise is equivalent to a viewing transformation that rotates the camera clockwise, for example. Finally, we know that the viewing transformation commands must be called before any modeling transformations are performed, so that the modeling transformations take effect on the objects first.

At the very beginning, two coordinate system, *eye coordinate* system and *object coordinate* system are the same. If we draw an object at this moment, it will be located at the *origin* of the eye coordinate system. When we apply model transformation, we change only the object coordinates. For example, we apply a translate transformation to the

object. The world coordinate does not change. But the object coordinates move to a particular point specified by translate command. As a result, the object finally appear at the point instead of the origin of the eye coordinate origin.

The point we look from is usually called the *eye position*, *viewing point* or *camera position*. The direction look into is called the *view direction*. The point that appears in the centre of the screen is called a *look-at point* or *model point*.

OpenGL function `gluLookAt()` is used to define a eye coordinate system. The call of this function is also a model view transformation. It encapsulates a series of rotation and translation commands, and form the modelview matrix. Thus it is a easy way to construct modelview matrix and does not require the programmer to do all these transformation step by step.

3.2.2.3 Projection Transformation

In addition to the field-of-view considerations, the projection transformation determines how objects are *projected* onto the screen, as its name suggests. OpenGL provides two basic types of projections, along with several corresponding commands for describing the relevant parameters in different ways. One type is the *perspective* projection, which matches how we see things in daily life. Perspective makes objects that are farther away appear smaller; for example, it makes railroad tracks appear to converge in the distance. If we are trying to make realistic pictures, we should choose perspective projection by using `glFrustum()` command or `gluPerspective()` command. The way these

two commands are used is as below:

```
glMatrixMode(GL_PROJECTION);  
glLoadIdentity();  
glFrustum(left, right, bottom, top, near, far);
```

This code creates a matrix for a perspective-view frustum and multiplies the current matrix by it. The frustum's viewing volume is defined by the parameters: (**left**, **bottom**, **-near**) and (**right**, **top**, **-near**) specify the (x, y, z) coordinates of the lower left and upper right corners of the near clipping plane; **near** and **far** give the distances from the viewpoint to the near and far clipping planes. They should always be positive.

A similar effect can be achieved in a different way by the following code:

```
glMatrixMode(GL_PROJECTION);  
glLoadIdentity();  
glPerspective(fovy, aspect, zNear, zFar);
```

This code creates a matrix for a symmetric perspective-view frustum and multiplies the current matrix by it. The **fovy** argument is the angle of the field of view in the x-z plane; its value must be in the range [0.0,180.0]. The **aspect** ratio is the width of the frustum divided by its height. The **zNear** and **zFar** values are the distances between the viewpoint and the clipping planes, along the negative z-axis. They should always be positive.

The other type of projection is orthographic, which maps objects directly onto the

screen without affecting their relative size. Unlike perspective projection, the size of the viewing volume of this kind of projection does not change from one end to the other, so distance from the camera does not affect how large an object appears. Orthographic projection is used in architectural and computer-aided design applications where the final image needs to reflect the measurements of objects rather than how they might look. Orthographic projection is set as default in OpenGL.

By default, viewing volume orthographic projection is the 2*2 cube where x is between $[-1, 1]$, y is between $[-1, 1]$, z is between $[-1, 1]$, and with the screen in the plane $z = 0$. An example:

```
glMatrixMode(GL_PROJECTION);  
glLoadIdentity();  
glOrtho(left, right, bottom, top, near, far);
```

The command `glOrtho()` creates an orthographic parallel viewing volume. It makes object display in a way as if we are viewing the window from an infinitely far point.

In our implementation, we deployed *perspective* projection.

3.2.2.4 Viewport Transformation

Together, the projection transformation and the viewport transformation determine how a scene gets mapped onto the computer screen. The projection transformation specifies the mechanics of how the mapping should occur, and the viewport indicates the shape of the available screen area into which the scene is mapped. Since the viewport specifies the

region the image occupies on the computer screen, we can think of the viewport transformation as defining the size and location of the final processed photograph - whether it should be enlarged or shrunk, for example.

The call

```
glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

defines a pixel rectangle in the window into which the final image is mapped. The (x, y) parameter specifies the lower left corner of the viewport, and **width** and **height** are the size of the viewport rectangle. By default, the initial viewport values are **(0, 0, winWidth, winHeight)**, where **winWidth** and **winHeight** are the size of the window.

The *aspect* ratio of a viewport should generally equal the *aspect* ratio of the viewing volume. If the two ratios are different, the projected image will be distorted as it is mapped to the viewport.

3.2.2.5 OpenGL Function **gluLookAt()**

The OpenGL function `gluLookAt()` is used to define the eye position, or camera position, the direction to look into, and the up direction (upward vector). Calling this function performs a modelview transformation since it changes the current modelview matrix. It makes the look-at point (model origin) separate away from the viewing point (eye coordinate origin) so that we can view the scene from arbitrary distance specified by

viewing point and lookat point.

Function `gluLookAt()` encapsulates a series of rotation and translation commands, and form the modelview matrix. It specifies a model-view transformation that simulates looking at the model or scene from a particular viewpoint.

```
void gluLookAt (eyeX , eyeY , eyeZ , centerX , centerY , centerZ ,  
upX , upY , upZ );
```

`eyeX, eyeY, eyeZ` - Specifies the position of the eye point.

`centerX, centerY, centerZ` - Specifies the position of the reference point.

`upX, upY, upZ` - Specifies the direction of the up vector.

`gluLookAt()` creates a viewing matrix derived from an eye point, a reference point indicating the center of the scene, and an UP vector.

The matrix maps the reference point to the negative z axis and the eye point to the origin. When a typical projection matrix is used, the center of the scene therefore maps to the center of the viewport. Similarly, the direction described by the UP vector projected onto the viewing plane is mapped to the positive y axis so that it points upward in the viewport. The UP vector must not be parallel to the line of sight from the eye point to the reference point. We will show how the modelview matrix is formed in the following description.

Let $N = (\text{centerX} - \text{eyeX}, \text{centerY} - \text{eyeY}, \text{centerZ} - \text{eyeZ})$

Let $V = (\text{upX} , \text{upY} , \text{upZ})$

Since vector N defines the direction for the Z_{view} axis direction and the view-up vector V is used to obtain the direction for the Y_{view} axis, we need only determine the direction for the X_{view} axis.

First, we can get the normal along the Z_{view} axis,

$$n = \frac{N}{|N|} = (n_x, n_y, n_z)$$

We can compute a third vector U that is perpendicular to both N and V . And we get u (u defines the direction for the positive X_{view} axis) by:

$$u = \frac{V \times n}{|V|} = (u_x, u_y, u_z)$$

The vector cross product of n and u also produces the adjusted value for V , perpendicular to both N and U , along the positive Y_{view} axis.

$$v = n \times u = (v_x, v_y, v_z)$$

Transformation from world to viewing coordinate, we need to:

- (1) Translate the viewing-coordinate origin to the origin of the world-coordinate system.
- (2) Apply rotations to align the X_{view} , Y_{view} , Z_{view} axes with the world X_w , Y_w and Z_w axes, respectively.

The viewing-coordinate origin is at world position $P = (eyeX, eyeY, eyeZ)$.

Therefore, the matrix for translating the viewing origin to the world origin is:

$$T = \begin{bmatrix} 1 & 0 & 0 & -eyeX \\ 0 & 1 & 0 & -eyeY \\ 0 & 0 & 1 & -eyeZ \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For the rotation transformation, we can use the *unit* vectors u , v , and n to form the composite rotation matrix that superimposes the viewing axes onto the world frame.

This transformation matrix is

$$R = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where the elements of matrix R are the components of the u v n axis vectors.

And `gluLookAt()` is equivalent to:

```
glMultMatrixf(R);
glTranslated(-eyex, -eyey, -eyez);
```

In the above description, we have $P = (eyeX, eyeY, eyeZ)$. The coordinate transformation matrix is then obtained as the product of the preceding translation and rotation matrices:

$$M_{wc,vc} = R.T = \begin{bmatrix} u_x & u_y & u_z & -u.P \\ v_x & v_y & v_z & -v.P \\ n_x & n_y & n_z & -n.P \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matrix $M_{wc,vc}$ transfers world-coordinate object descriptions to the viewing reference frame. Translation factors in this matrix are calculated as the vector dot product of each of the u, v and n unit vectors with P , which represents a vector from the world origin to the viewing origin. In other words, the translation factors are the negative projections of P on each of the viewing-coordinate axes (the negative components of P in the viewing coordinates). These matrix elements are evaluated as

$$-u.P = -eyeX * u_x - eyeY * u_y - eyeZ * u_z$$

$$-v.P = -eyeX * v_x - eyeY * v_y - eyeZ * v_z$$

$$-n.P = -eyeX * n_x - eyeY * n_y - eyeZ * n_z$$

From the above analysis, we know `gluLookAt()` constructs a modelview matrix automatically, and avoids the programmer having to do these calculation step by step. Clearly, it is much easier to use `gluLookAt()` than to perform these calculations in the application code. So we are going to use OpenGL **`gluLookat()`** to implement our camera control.

Chapter 4 Design

In this thesis, We propose methods for obtaining cinematic camera positions and provide a camera module that can be used to efficiently and effectively direct camera control in computer graphics game animation, presentation or computer animated cartoons. We define a camera class and provide a series of functions that can be used to control camera movement, camera collision detection and cinematic camera position calculation. Also, we will apply this camera module to a room conversation animation between two people to demonstrate the effectiveness and efficiency of the camera control.

In order to obtain the objective described above, we will separate our design into two parts. First we will describe the camera module and especially the cinematic camera position calculating algorithm design. Then we will describe the room conversation animation implemented by cinematic camera control.

4.1 Camera module design

4.1.1 Camera components

The design of camera module is showed in figure 4-1. It has three components: camera movement, collision detection, and position calculation. In order to make this module more general, I will use vector, quaternion and matrix transformations. The camera can

move left/right, up/down, forward/backward, pan and tilt. Whenever a viewer or player moves the camera in the virtual environment, collision detection is necessary in order to avoid camera collision with any objects in the environment. A camera cinematic position calculation analyze people's position in the environment, camera up direction, and user input parameters or default value of those parameters, to obtain an appropriate camera position and model position to look at. Thus the viewer or player will see cinematic visual pictures on the screen.

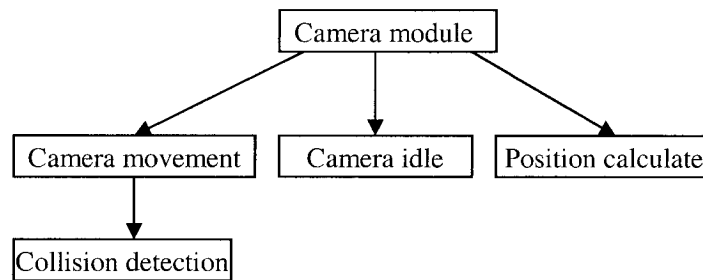


Figure 4.1 Camera Components

Camera movement includes: move left/right, move up/down, move forward/backward, pan and tilt. In order to make the camera movement function general, we will use matrix and quaternion. Camera idle is used to make camera movement smooth.

For collision detection, we must calculate an object's *bounding box*. In 3D graphics environment, the bounding box of an object has six faces. We build a *plane equation* for each *face*, specify its *front* and *back* face direction. And we can imagine that the camera is a sphere which has a radius r . Normally, when a camera moves to position point, we calculate the distance from the camera sphere center to each plane of an object's

bounding box. The distance value could be positive or negative. If its absolute value is less than the radius r of the sphere, then the camera intersects with the plane, implying that a collision occurs. If the distance is greater or equal to the r , then the camera is in front of the plane. When camera is in front of any one plane of a bounding box, we do not need to detect other planes. And we know that no collision happens. If the distance is negative and its absolute value is greater than r , then camera is at the *back* of a plane of the bounding box. We must check if the camera is at the *back* of all the six planes of the bounding box. If it is, it means camera collides with the object.

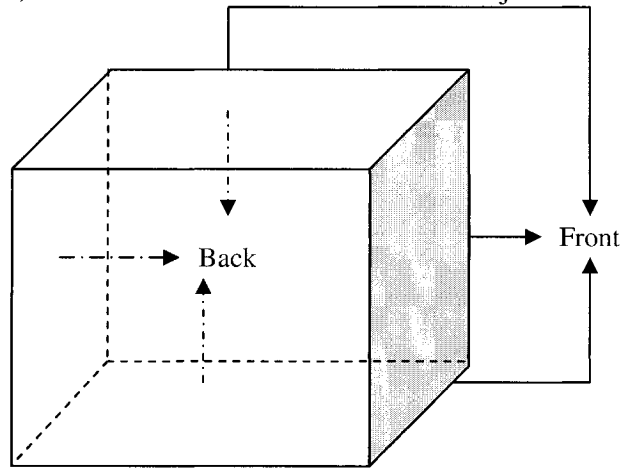


Figure 4.2 Bounding Box for objects inside a room

To check if camera collides with a room, the criteria is opposite to that of objects in the room. We treat inside direction of a room as *front* and outside direction as *back*. If camera is behind or intersects with one plane of a room's bounding box, then it collides with the room, and we do not bother to check other planes. The same reason, if camera is in front of all the planes of a room's bounding box, then we conclude that camera is inside the room and no collision occurs.

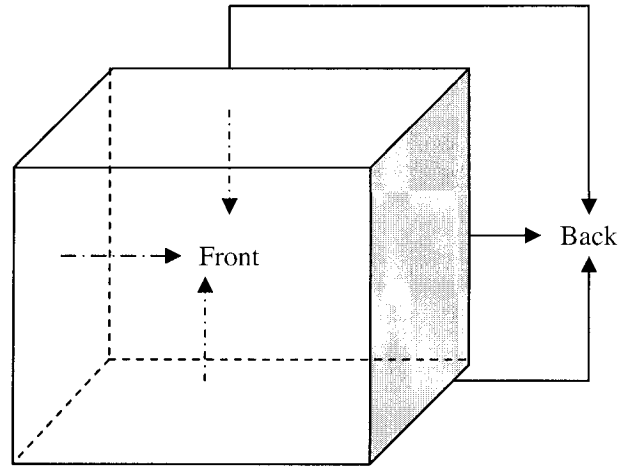


Figure 4.3 Bounding Box for a room

4.1.2 Cinematic camera position algorithm design

In chapter 3, We introduced the basic concepts of film grammar. In general, the position of the camera should be based on conventional techniques that have been established in filming a conversation. The placement of the camera is based on the position of the two people having the conversation. Here, we will define constraints for an over-the-shoulder shot and constraints for a corresponding over the shoulder shot.

Constraints for an over the shoulder shot [Drucker 95]

- The height of the character facing the view should be approximately $1/2$ the size of the frame.
- The person facing the view should be at about the $2/3$ line on the screen.
- The person facing away should be at about the $1/3$ line on the screen.
- The camera should be aligned with the world up.
- The field of view should be between 20 and 60 degrees.

- The camera view should be as close to facing directly on to the character facing the viewer as possible.

Constraints for a corresponding over the shoulder shot:

- The same constraints as described above apply here, but the people should not switch sides on the screen; therefore the person facing towards the screen should be placed at the 1/3 line and the person facing away should be placed at the 2/3 line.

After analyzing the above constraints, we can keep the camera up vector always aligned with world up vector, make camera look at the head of those people in order to make them appear 1/2 the size of the screen, and limit the camera perspective angle between 20 and 60 degrees. To calculate the camera position in order to satisfy other specifications above, we will analyze the algorithm in the below descriptions.

In OpenGL coordinate system or world coordinate system: Y is up; X is to the right; and Z is towards the camera (eye). According to chapter 3, film grammar assigns great importance to the players' heads, and almost always requires the camera to point at them. In our algorithm, we will apply this rule.

There are two people (i.e., heads): a nearer one at N and a further one at F . F is the person we want to look at. The camera is at C . We are interested in the triangle CNF . Note that n is the distance from the camera to the nearer head and f is the distance to the further head.

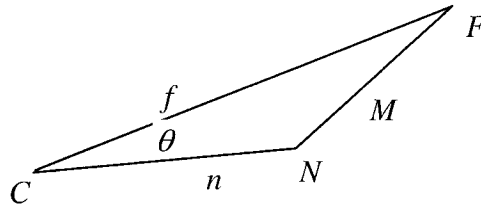


Figure 4.4 Triangle formed by two players and the camera

Since three points define a plane, CNF is a plane. The first thing we compute is the angle θ at C : this is the angle between the two heads as seen from the camera. Since according to the film grammar, our constraints are, one head should appear at position $\frac{1}{3}$ and the other at position $\frac{5}{6}$ on the screen.

Using a perspective projection, we have the Y angle β (called *fovy* in OpenGL) and the aspect ratio $r = w/h$. The relation between the X angle α and the Y angle β is $\frac{\tan(\alpha/2)}{\tan(\beta/2)}$, and $r = \frac{\tan(\alpha/2)}{\tan(\beta/2)}$. Consequently

$$\alpha = 2 \tan^{-1}(r \tan(\beta/2))$$

A perspective projection frustum volume of CNF is showed below:

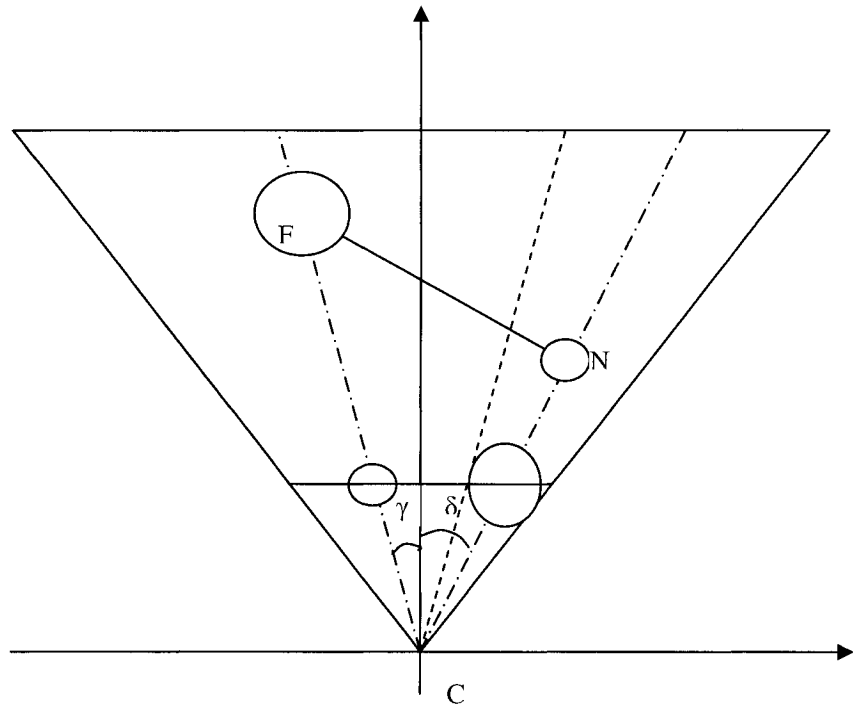


Figure 4.5 Perspective projection frustum of two players and camera

From Figure 4.3, we can see:

$$\tan(\gamma) = \left(\frac{d}{3} * \tan(\alpha/2)\right) / d = \frac{\tan(\alpha/2)}{3}$$

$$\tan(\delta) = \left(\frac{2d}{3} * \tan(\alpha/2)\right) / d = \frac{2 \tan(\alpha/2)}{3}$$

So

$$\gamma = \tan^{-1}\left(\frac{1}{3} \tan(\alpha/2)\right)$$

$$\delta = \tan^{-1}\left(\frac{2}{3} \tan(\alpha/2)\right)$$

Then,

$$\theta = \gamma + \delta$$

We can extend this idea, applying it to display the further person on $\frac{m}{l}$ of the screen, and nearer person on $\frac{l-m}{l}$ of the screen. Then

$$\tan(\gamma) = ((1 - \frac{m}{l})d * \tan(\alpha/2)) / d = (1 - \frac{m}{l}) \tan(\alpha/2)$$

$$\tan(\delta) = (\frac{m}{l}d * \tan(\alpha/2)) / d = \frac{m}{l} \tan(\alpha/2)$$

So

$$\gamma = \tan^{-1}((1 - \frac{m}{l}) \tan(\alpha/2))$$

$$\delta = \tan^{-1}(\frac{m}{l} \tan(\alpha/2))$$

By the same reasoning,

$$\theta = \gamma + \delta$$

Applying the cosine rule to the triangle CNF gives

$$c^2 = f^2 + n^2 - 2fn \cos \theta$$

Assume that the further head F is k times as far from the camera as the nearer head

N . For most applications, k is roughly between 3 and 4 in an over the shoulder shot.

Then $f = kn$ and we have

$$\begin{aligned} c^2 &= k^2 n^2 + n^2 - 2fn \cos \theta \\ &= n^2 (k^2 + 1 - 2k \cos \theta) \end{aligned}$$

Let

$$s^2 = k^2 + 1 - 2k \cos \theta$$

Note that s is a constant that depends only on the perspective projection and the value

chosen for k . We also know c , because it is the distance between the heads. We now

have:

$$n = c/s$$

$$f = kc/s$$

Next, assign coordinates in the CNF plane to each point:

$$C \equiv (x_c, y_c, z_c)$$

$$N \equiv (x_n, y_n, z_n)$$

$$F \equiv (x_f, y_f, z_f)$$

so that

$$(x_c - x_n)^2 + (y_c - y_n)^2 + (z_c - z_n)^2 = n^2$$

$$(x_c - x_f)^2 + (y_c - y_f)^2 + (z_c - z_f)^2 = f^2$$

Then we have

$$(x_c - x_n)^2 + (y_c - y_n)^2 + (z_c - z_n)^2 = c^2 / s^2$$

$$(x_c - x_f)^2 + (y_c - y_f)^2 + (z_c - z_f)^2 = k^2 c^2 / s^2$$

We have to solve these equations to find the camera position (x_c, y_c, z_c) . To simplify the equations, put the near person at $(0,0,0)$ and the far person at $(1,0,0)$. Then

$$x_n = 0$$

$$y_n = 0$$

$$z_n = 0$$

$$x_f = 1$$

$$y_f = 0$$

$$z_f = 0$$

In this simplified coordinate system, we have $c = 1$. With these substitutions, the equations become to the equations below:

$$x_c^2 + y_c^2 + z_c^2 = 1/s^2 \quad (1)$$

$$(x_c - 1)^2 + y_c^2 + z_c^2 = k^2 / s^2 \quad (2)$$

Subtracting (1) from (2), eliminates z_c and gives

$$(x_c - 1)^2 - x_c^2 = (k^2 - 1)/s^2$$

which we can solve for $2x_c$ giving

$$\begin{aligned} 2x_c &= 1 - \frac{k^2 - 1}{s^2} \\ &= \frac{s^2 - k^2 + 1}{s^2} \\ &= \frac{k^2 + 1 - 2k \cos \theta - k^2 + 1}{s^2} \quad (\text{using } s^2 = k^2 + 1 - 2k \cos \theta) \\ &= 2 \left(\frac{1 - k \cos \theta}{s^2} \right) \end{aligned}$$

and so

$$x_c = \frac{1 - k \cos \theta}{s^2}$$

Substitute x_c into equation (1), we have:

$$\begin{aligned} y_c^2 + z_c^2 &= \frac{1}{s^2} - x_c^2 \\ &= \frac{1}{s^2} - \frac{(1 - k \cos \theta)^2}{s^4} \\ &= \frac{k^2 + 1 - 2k \cos \theta - 1 + 2k \cos \theta - k^2 \cos^2 \theta}{s^4} \\ &= \frac{k^2 \sin^2 \theta}{s^4} \end{aligned}$$

and therefore, we can get

$$\begin{aligned} y_c &= \pm \frac{k \sin \theta}{s^2} \cdot \cos \sigma \\ z_c &= \pm \frac{k \sin \theta}{s^2} \cdot \sin \sigma \quad (\sigma \geq 0^\circ \text{ and } \sigma \leq 360^\circ) \end{aligned}$$

The positive and negative square roots correspond to two possible camera positions. The diagram shows one position; the other position is obtained by reflecting C in the line NF in figure 4.2.

We now have the camera position (x_c, y_c, z_c) , in the special coordinate system. To obtain the true camera position in the original coordinate system, we apply the following transformations to (x_c, y_c, z_c) :

1. Rotate about the Y axis through an angle ϕ where

$$\cos \phi = \frac{x_f - x_n}{d_1}$$

$$\sin \phi = \frac{z_f - z_n}{d_1}$$

$$\text{where } d_1 = \sqrt{(x_f - x_n)^2 + (z_f - z_n)^2}$$

2. Rotate about the X axis through an angle φ , where

$$\cos \varphi = \frac{y_f - y_n}{d_2}$$

$$\sin \varphi = \frac{z_f - z_n}{d_2}$$

$$\text{where } d_2 = \sqrt{(y_f - y_n)^2 + (z_f - z_n)^2}$$

3. Scale (increase the distance NF from 1 to c , and other distances in proportion, where

$$c = \sqrt{(x_f - x_n)^2 + (y_f - y_n)^2 + (z_f - z_n)^2}.$$

$$x' = cx$$

$$y' = cy$$

$$z' = cz$$

4. Translate (move N to its correct position (x_n, y_n, z_n)),

$$x' = cx + x_n$$

$$y' = cy + y_n$$

$$z' = cz + z_n$$

When these transformations have been applied to (x_c, y_c, z_c) , we should have the correct camera position.

As a check, the same transformations applied to $(0, 0, 0)$ and $(1, 0, 0)$ should give the correct positions for $N \equiv (x_n, y_n, z_n)$ and $F \equiv (x_f, y_f, z_f)$, respectively. M is a point on the line of CF rotate about Y axis $-\gamma$ get F' , or CN rotate about Y axis δ get N' . Here, we treat $M = \frac{F' + N'}{2}$ to make it more accurate. In OpenGL, the look-at point is at Z axis of the view coordinate system. So we can extend M along Z view axis to obtain a more distant model point, then we have less trouble to do camera pan and tilt. In our program, we use $M' = M + (M - C) * 30$ to replace M . This can give us a more distant model point and the (x, y, z) coordinate value of M are restricted not to be an infinite number.

Then we use C as the eye position and M' as the model position in the `gluLookAt()` call.

As a first extension of the algorithm, if we want to look at a person from another person's view (an internal reverse camera placement), which implies that the person being looked at occupies the whole screen, then camera C should be along the line NF and between point N and F . Thus we have:

$$\frac{CF}{CN} = k, \text{ and}$$

$$CF + CN = c$$

Thus,

$$CF = \frac{k}{k+1}c$$

$$CN = \frac{1}{k+1}c$$

Different k will give different camera position C . And the corresponding model point M should be the point which the person being looked at.

Sometimes we can treat the nearer person N as the object we want to look at (for not an over the shoulder shot), the same reasoning from above analysis, we have:

$$\tan(\gamma) = (\frac{m}{l}d * \tan(\alpha/2)) / d = \frac{m}{l} \tan(\alpha/2)$$

$$\tan(\delta) = ((1 - \frac{m}{l})d * \tan(\alpha/2)) / d = (1 - \frac{m}{l}) \tan(\alpha/2)$$

So

$$\gamma = \tan^{-1}(\frac{m}{l} \tan(\alpha/2))$$

$$\delta = \tan^{-1}((1 - \frac{m}{l}) \tan(\alpha/2))$$

M is a point on the line of CN rotate about Y axis $-\gamma$ get N' , or CF rotate about Y axis δ get F' . Because N' and F' are along the view Z axis, and the model point M is along the view Z axis, too. We don't need to know the exact coordinate value of M . So here, we choose an average value of N' and F' , calculate $M = \frac{F' + N'}{2}$. In OpenGL, the look at point is at Z axis of the view coordinate system.

So we can extend M along the Z axis to obtain a more distant model point, then we have fewer problems when we pan or tilt the camera. In our program, the same reason as

before to limit the coordinate value of M , we use $M' = M + (M - C) * 30$ to replace M .

As a second extension of the algorithm, when one person is walking, we want to put him/her on the specified position on the screen, according to the analysis above, the same reason, we put model position M at a position that the line and the view Z axis form an angle of γ , and

$$\gamma = \tan^{-1}\left(\frac{m}{l} \tan(\alpha/2)\right)$$

4.2 Design of an Animated Conversation

4.2.1 Project Components

The main components of the project are displayed in figure below:

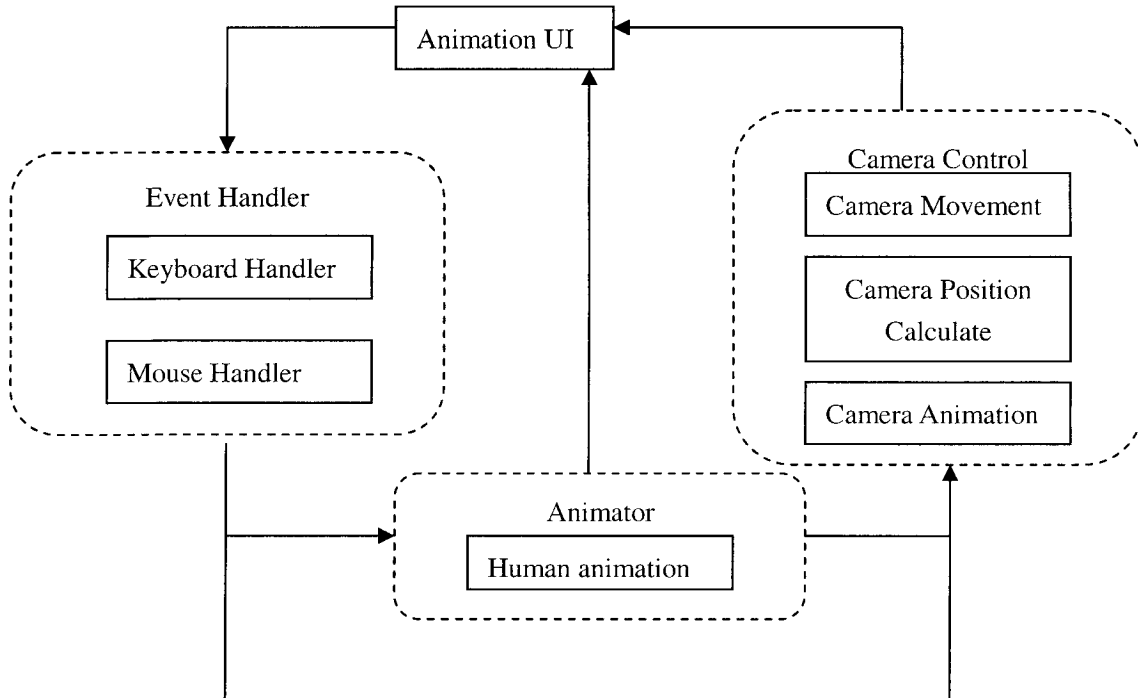


Figure 4.6 Program Components

The animation program is composed of four parts. 1) A user interface component allows users to interact with the program, 2) An event handler that can accept user input from keyboard or mouse click and notify the camera animation or human action animation, 3) A human animation component that accepts the message from event handler and control the people action, and decides which action a person should taken. Human action such as walking can cause the camera to following a person, and 4) A camera control component that accepts the message from event handler to control camera movement or to calculate camera position.

4.2.2 Film scenes to be animated

The design of the animated conversation demonstrates the correctness of the previous camera control and cinematic camera position algorithms. This animation will consist of 5 scenes.

- 1) Two people sit down having a conversation, while the camera switches from one person to another.
- 2) One person stands up while another one sits.
 - a) The same animation as in 1).
 - b) The camera looks at a person from another person's viewpoint.
- 3) One person sits down while another one walks toward him/her
 - a) Using 3 shots to describe this scene.
 - b) Using 4 shots to describe this scene.

Chap 5 Implementation

The entire code of this project was written by me in C++ and OpenGL except the CUGL (Concordia University Graphics Library) <http://www.cs.concordia.ca/~grogono/CUGL>. We built a camera module based on the CUGL camera class.

As mentioned in Chapter 3, OpenGL provides a rich graphics API. It consists of 3 libraries: 1) the Graphics Library (GL), 2) the Graphics Library Utilities (GLU), 3) and the Graphics Library Utilities Toolkit (GLUT). For the program user interface, we used the GLUT-based C++ user interface library (GLUI).

The project is implemented by rendering a room scene with two people sitting on opposite sides of a table. Users can interact with the program to control the camera or control the peoples' actions using keyboard commands or mouse clicks. Users can also input parameters on the screen panel in order to get the desired cinematic visual effects. The program gives the effect of a film of two people having a conversation. During the conversation we may have the following scenarios: 1) both of the people may be sitting down; 2) one sits while the other one stands; 3) one sits while the other one walks; etc. The camera can move left/right, up/down, forward/backward with collision detection, or pan and tilt.

5.1 User interface implementation

The program interface implementation is divided into two parts. The display window shows the whole animation scene, the right and bottom of the window frame are panels with parameter edit boxes and buttons to enable users to interact with the program. Users can also interact with the program from the keyboard. Every time the user inputs some data or clicks a button, the animation scene will change as the user desired.

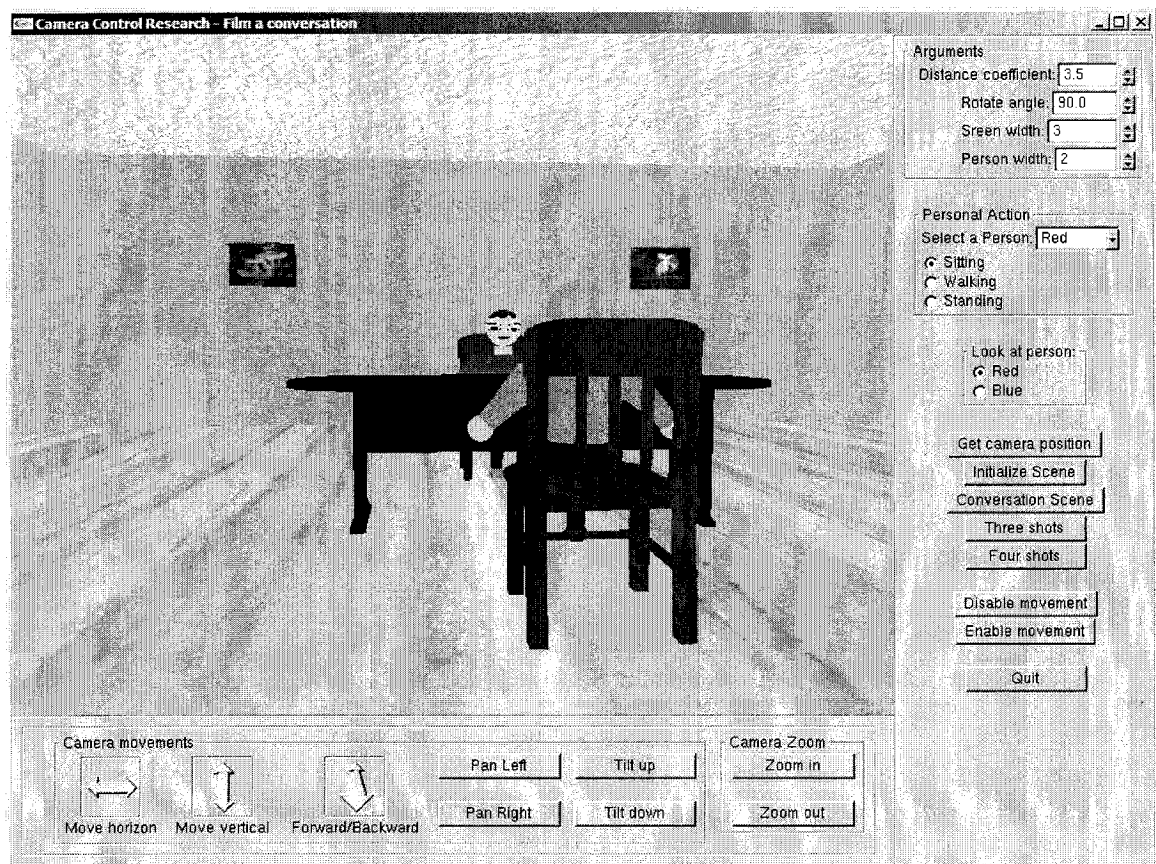


Figure 5.1 Program interface

Components on the right panel provide control for personal actions, camera position calculations, or camera animation.

Distance Coefficient

The *distance coefficient* is an argument that was discussed in Chapter 4, the distance scale of CF by CN, we denote it k here to simplify the explanation. The value of this argument reflects the camera position relative to both the nearer person and further person.

The value of the coefficient k determines the distance of the camera to one of the people in the scene. A value close to 0 may place the camera too close to the nearer person's head, blocking the view of the further player. A value of 1 indicates that the camera is at the apex position. For conversational scenes, we have found that values from 0 to 10 are appropriate.

Rotation Angle

Changing the value of the *rotation angle* (corresponds to the angle σ introduced in chapter 4) affects the camera position to look at players. It will affect the peoples' position displayed on the screen. It will make the camera rotate along the line of interest between the players. This angle could be between 0° and 360° , and values within some extents between 0° and 360° are really effect, not all the values between 0° and 360° can get the correct camera position because they either will collide with objects in the room space or in a position that are relative too low or too high to the players' heads.

Screen Width to Person Width Ratio

Usually we give a numerator and a denominator to indicate the ratio of person space to total screen space. The ratio is usually specified according to film grammar. An example is:

- The person facing the view should occupy about 2/3 line of the screen.
- The person facing away should occupy about 1/3 of the screen

Personal Action

By choosing a person from the combo box, and clicking on the **sit, stand** or **walk** button, users can control a person's action by making him/her sit down, stand up or walk.

Looking at a Person

By clicking on the **red** or **blue** button, users can make the camera look at the desired person. The selected person will face the camera.

Get Camera Position

Whenever a user has finished parameter input or make a choice among the radio button, clicking the **Get Camera Position** button will make the program recalculate the camera position and display the corresponding scene.

Initialize Scene

Click **Initialize Scene** button, the program will arrange players **Red** and **Blue** to sit around the table as the initial program scene.

Conversation Scene

Click **Conversation Scene** button, the program will automatically display the conversation scene between **Red** and **Blue** when they are sitting and facing each other. This scene we have described in Chapter 3.

Three shots

Click **Three shots** button, the program will automatically play the three-shots scene that **Red** is approaching **Blue**.

Four shots

Click **Four shots** button, the program will automatically play the four-shots scene that **Red** is approaching **Blue**.

Enable/Disable bottom panel button

This button enables or disables buttons on the bottom panel. The buttons on the bottom panel are camera movement control buttons.

Move left/right

Clicking the mouse left button on this button icon and keeping the mouse pressed while moving left or right, causes the camera to move to left or right. Releasing the mouse button finishes the camera movement.

Move up/down

Clicking the mouse left button on this button icon and keeping the mouse pressed while moving up or down, causes the camera to move to up or down in the scene. Releasing the mouse button finishes the camera movement.

Move forward/backward

Clicking the mouse left button on this button icon and keeping the mouse pressed and move up or down, causes the camera to move forward or backward in the scene.

Releasing the mouse button finishes the camera movement.

Pan left/right button

Clicking the mouse left button on pan left button icon makes the camera pan to the left in the scene; clicking the mouse on pan right button icon makes the camera pan to the right in the scene. Releasing the mouse button finishes the camera movement.

Tilt up/down

Clicking the mouse left button on tilt up button icon makes the camera tilt up in the scene; clicking the mouse on tilt down icon makes the camera tilt down in the scene. Releasing the mouse button finishes the camera movement.

Zoom in/Zoom out

Clicking the mouse left button on this button zooms the view in or out.

5.2 Camera algorithm Implementation

The camera is defined as a class. Its behavior/interface consists of the following functions: move left/right, up/down, forward/backward, tilt and pan. It also has functions to detect collisions and to get the specified camera position according to user input parameters.

5.2.1 Data structure

The data structure of a camera class is shown below:

```
enum CamPosToPlane  
{ INTERSECTS, FRONT, BEHIND };
```

```

class Camera
{
public:
    // Set camera position, look at position and up vector
    void set(const Point &eye, const Point &model, const Vector &up);
    void set(const Point &eye, const Point &model);
    ...
    // Update the camera position.
    // This function should be called from the GLUT idle() function
    // or its equivalent. It performs one step of the motion until the motion is complete
    void idle();
    void apply(); // Apply the camera position to the model-view matrix.

    void moveUp(GLfloat distance);
    void moveDown(GLfloat distance);
    void moveForward(GLfloat distance);
    void moveBackward(GLfloat distance);
    void moveLeft(GLfloat distance);
    void moveRight(GLfloat distance);
    void Tilt(unsigned char dir);
    void Pan(unsigned char dir);

    void setRadius(float r); // Set the collision detect radius for a camera

    // check if camera collide with a object
    CamPosToPlane ClassifyCamPos(PLANE pl, float& distance);
    bool CollideWithPlane(PLANE pl, float& distance, int flag);
    bool DetectCollision(float& distance);

    ...
    // calculate desired camera position according to the person we look at
    Point Calc_Model_Pos(Point n, Point f, int l, int m, Point cam,
                        int flag_m, int flag_c);
    Point Calc_Cam_Pos(Point n, Point f, int l, int m, float
                      dist_coeff, float yz_rot_angle, int flag);
    Point Walk_Model_Pos(Point f, int l, int m, Point cam);

private:
    Point eye; // Eye coordinates for gluLookAt()
    Point model; // Model coordinates for gluLookAt()

```



```

Vector up;      // Up vector for gluLookAt()

int  steps;    // Current value of step counter
int  maxSteps; // Maximum number of steps for a smooth movement

float fRadius; // The camera's collision radius
...
};

```

The camera class's member variables **eye**, **model**, and **up** correspond to the camera position, the model position and the up vector, respectively. The variable **fRadius** is the radius of the camera sphere. Here, we treat camera as a sphere. Whenever we want to know if the camera collides with any objects, we just calculate the distance from the centre of the sphere to the objects' bounding boxes plane, and in which direction the camera sphere locate relative to the bounding box planes. Variable **steps** and **maxSteps** are used for camera idle() function for film animation, by setting the camera and model position change step, we make the view scene changing very smoothly.

The effect of the functions **moveUp**, **moveDown**, **moveForward**, **moveBackward**, **moveLeft**, **moveRight**, **Tilt**, **Pan**, is obvious from their names.

Function **ClassifyCamPos()** can tell the relative position between the camera and the objects' bounding box. It could be *Intersects*, *Front* or *Back*, which means the camera could collide with an object, in front of an object or in the back of the bounding box plane of an object.

Function **CollideWithPlane()** is used to tell if the camera collides with an

object. And **DetectCollision()** is a general function to check all the objects in the graphics scene, if the camera collides with any of them. So it is convenient for the programmer to simply the collision detect code, just use one line to call camera **DetectCollision()**, and it can tell if there is any collision.

The most important functions in this thesis are **Calc_Model_Pos()**, **Calc_Cam_Pos()** and **Walk_Model_Pos()**. **Calc_Cam_Pos()** will calculate the desired camera position according to the persons' coordinate in the scene. **Calc_Model_Pos()** can be called to get the model position the camera looking at. And Function **Walk_Model_Pos()** are used to get the model position the camera looking at when one person is walking.

5.2.2 Algorithms Implementation

5.2.2.1 Camera Movement

- Move left/right

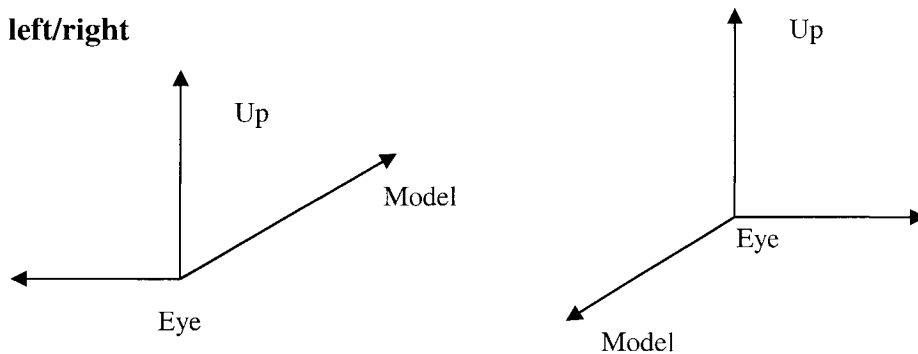


Figure 5.2 Cross product of Up and model - eye or eye - model

Move left: $\text{Vector disp} = \text{distance} * \text{cross}(\text{up}, \text{model} - \text{eye}).\text{unit}();$

Move right: $\text{Vector disp} = \text{distance} * \text{cross}(\text{up}, \text{eye} - \text{model}).\text{unit}();$

The points **model** and **eye** give the positions of the point of interest and the eye, or camera, respectively. Their difference is a vector representing the displacement from the model to the view. The vector **up** points vertically upwards in the mode. The cross-product of these vectors is a vector that is horizontal with respect to the model and perpendicular to the direction of the camera. Consequently, it is the appropriate direction for a camera movement to the left or right. The direction is chosen according to the sign of this vector: **model - eye** or **eye - model**. The function **unit()** converts this vector to a unit vector in the same direction and the scalar multiple **distance** gives an appropriate distance for the movement.

□ **Move up/down**

Move up: $\text{Vector disp} = \text{distance} * \text{up};$

Move down: $\text{Vector disp} = \text{distance} * (-\text{up});$

Camera **Up** vector is a unit vector, multiple it with a given distance argument, then we get the camera movement offset along the model view Y axis. When multiplying **Up** vector with the distance argument, the camera move up along the Y axis. Multiplying **-Up** vector with the distance argument, the camera move down along the Y axis. Both the camera and model position are changed by movement offset.

□ **Move forward/backward**

Move forward: `Vector disp = distance * (model - eye).unit();`

Move backward: `Vector disp = distance * (eye - model).unit();`

Looking from camera position to the model direction is the Z axis in the model view coordinate system. Get this vector and make it unit, multiply with the distance argument, we have the camera movement offset to move the camera forward. Get the negative of this vector, multiply the **distance** argument, we can move the camera backward. Both the camera and model position has changed by movement offset.

□ **Tilt**

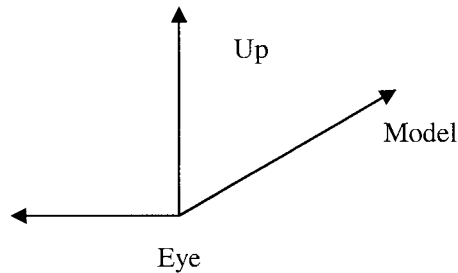


Figure 5.3 Cross product of Up and model-eye

`Vector axis = cross(eye - model, up).unit();`

`Quaternion rot(axis, angle/180.f*PI);`

`update(eye, rot.apply(model));`

This is implemented by keeping the *camera (eye)* position still while make the model position rotate around X axis in the eye coordinate system. As mentioned above, get the unit vector along eye coordinate system X axis. Use this vector to make a rotate

quaternion, that is, rotate the **model** point around this X axis about an **angle** (if angle is negative, then rotate in a clockwise order (tilt down); if it's positive, then rotate in a counter-clockwise order (tilt up)). Make a rotation matrix according to this quaternion, and multiply this matrix with the **model** point, then we get the destination model position we want to tilt the camera to.

□ **Pan**



Figure 5.4 Camera rotate around Up vector

```
Quaternion rot(up, angle/180*PI);  
  
update(eye, rot.apply(model));
```

This is implemented by keeping the *camera (eye)* position still while make the **model** position rotate around Y axis in the eye coordinate system. Use camera **Up** vector to make a rotate quaternion, that is, rotate the **model** point around this **Up** vector about an **angle** (if angle is negative, then rotate in a clockwise order-pan right; if it's positive, then rotate in a counter-clockwise order-pan left. Make a rotation matrix according to this quaternion, and multiply this matrix with the model point, then we get the destination model position we want to pan the camera to.

5.2.2.2 Collision detection

In chapter 4, we have described the principle of collision detect. A source code for detect objects inside a room is listed below:

```
//////////////////////////////////////////////////////////////////
// Parameter: side[] – An array of class Side objects, a bounding box has six side
//           distance – Distance from a camera center to a bounding box plane
//           flag – 0: indicates an object inside a room, other: indicates a room
bool Camera::CheckCollision(Side side[],float &distance,int
flag)
{
    bool flag = true;

    // check collision status with every plane of the 6 surface around a Bounding Box
    for (int i = 0; i < 6; i++)
    {
        PLANE p1 = chair.m_Side[i].m_Plane; // get each plane
        float distance = 0;

        CamPosToPlane pos= ClassifyCamPos(p1, distance);

        // For an object inside a room, if camera is in front of any one of the 6 plane,
        // then we know it's not inside the bounding box, so we don't need detect
        // other planes further. Only if the camera is intersected or behind all the
        // planes, then we can determine it's intersected with the Bound Box
        if (flag == 0)
            if (pos == FRONT)
                return false;

        // For a room, if camera is intersect with or behind any one of the 6 plane, then
        // we know it's going too far, and we can stop detecting any other planes further
        else
            if (pos == INTERSECTS || pos == BEHIND)
                return true;
    }

    return flag;
}
```

We must check camera with every object inside the room and the room itself to determine

if a collision happens. If a collision really happens, the camera stops moving.

5.2.2.3 Get an ideal camera position according to user input parameters

According to the design analysis of camera position calculating algorithm in chapter 4, given two people coordinate position in a conversation scene, and camera **Up** should be aligned with the world up, I must get the appropriate camera position and a model position. Then we can use these two points to set the camera 6 DOFs, means **model** and **eye** point, and camera up aligned with the world coordinate up direction. The algorithms implemented can always get an ideal camera perspective view to satisfy the cinematography requirement. And by adjusting the value of those **distance coefficient**, **angle** and scale value, we can get a range of different camera, model positions to satisfy the film grammar. Below is an example of how to use the camera position calculating function to get the camera and model coordinate, and set the camera in a 3D environment. It is very easy to call the function.

```
Point eye, model, n, f;
```

```
// Get the person's coordinate who is backing to camera
```

```
n = NearPerson->GetCurPos();
```

```
// Get the person's coordinate who is facing to camera
```

```
f = FarPerson->GetCurPos();
```

```
// Get the desired camera position
```

```
eye = camera.Cal_Cam_Pos(n, f, L, M, dist_coeff, yz_rot_angle, flag_c);
```

```
// Get the model position which camera is looking at
model = camera.Calc_Model_Pos(n, f, L, M, eye, flag_m, flag_c);

// Set camera position and model position
camera.set(eye, model);
```

5.3 Program implementation

We use C++ and OpenGL to implement the source codes of all the objects in the project.

Pictures of some the objects rendered in this project is in Figure 5.7.

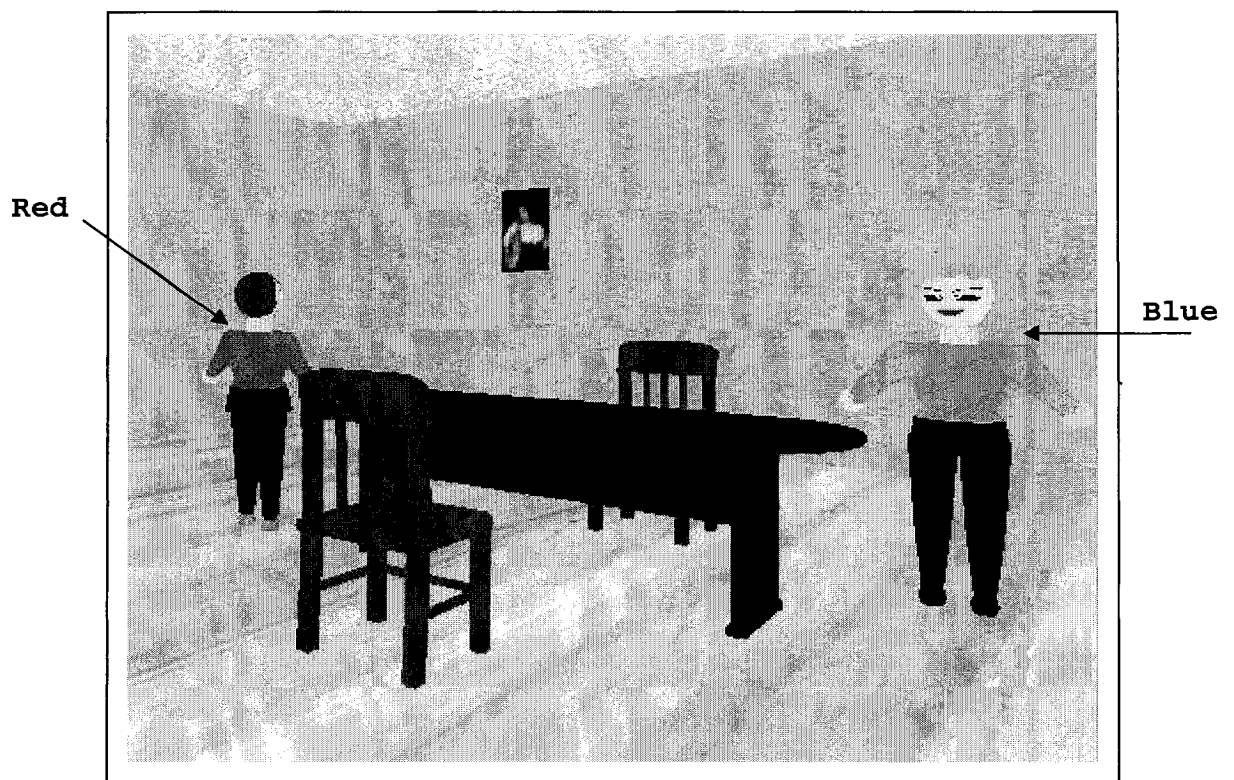


Figure 5.5 Animated Room Scene

5.3.1 Human being implementation

Human beings are rendered in a simple but natural way. For the purpose of this program, a human being is composed of a head, a neck, a body, left arm and right arm, left leg and

right leg, left hand and right hand, a hip, left foot and right foot. An arm is composed of an upper arm and a fore arm. An leg is composed of a upper leg and a lower leg. Organizing a human being in this way makes it easy to manipulate its action to make it sit down, stand up or walk. The corresponding classes are listed below:

```

class Person
{
public:
    Person();
    virtual ~Person();

private:
    ...
public:
    Side m_Side[6]; // 6 planes of Bounding Box

public:
    Head *head;
    Neck *neck;
    Body *body;
    UpperArm *leftUArm, *rightUArm;
    ForeArm *leftFArm, *rightFArm;
    Crus *leftCrus, *rightCrus;
    Femoral *leftFemo, *rightFemo;
    Hand *leftHand, *rightHand;
    Foot *leftFoot, *rightFoot;
    Hip *hip;

public:

    void Render(int headindex, int actionIndex);

    // Calculating the bounding box
    void Calc_Stand_Bound_Box();
    void Calc_Sit_Bound_Box();
    void Calc_Walk_Bound_Box();

```

```

    // Calculating the 6 plane equation of Bounding Box
    void Calc_Plane_Equation();
    ...
};
class Head
{
public:
    Head();
    virtual ~Head();

    ...
public:
    virtual void    Render(int index);
    ...
};

class Neck
{
public:
    Neck();
    virtual ~Neck();
    ...

public:
    virtual void    Render();
    ...
};

```

There are also **body**, **upper arm**, **fore arm**, **hand**, **hip**, **femoral** (upper leg), **crus** (lower leg), **foot** classes, which we choose not to show here because they will occupy too much space. Each person object is composed of all the above class objects from **head**, **neck**, etc. till **foot** classes. A **person** object will call the rendering functions of each body part independently, to render a **person** object. This way makes it easy to do human being sit, stand and walk animation.

5.3.1.1 Sit down animation

A human being can be made to sit down by rotating the hip through 90 degrees and rotating the lower leg 90 degrees in the opposite direction. The sitting person is then rendered into one display list. Whenever a standing person must be displayed, this display list is called.

5.3.1.2 Stand animation

By rendering the standing person into one display list, the hip and lower leg do not need to be rotated. Whenever we display the person standing, the stand display list is called.

5.3.1.3 Walk animation

This is implemented by alternating to display between two display list. One display list is rendered by make the human being left arm lift in front of the body while the right arm lift at back of the body, and the left femoral lift back of the body while the right femoral lift in front of the body. Another display list make arms and legs move opposite to the previous list. These display lists are called alternately to render a human walking.

5.3.2 Furniture (table and chair) implementation

Vertex data of table and chairs in this program are downloaded from 3DS files. A table and a chair 3DS files were downloaded from 3D CAFE website (<http://www.3dcafe.com/asp/househld.asp>). And the corresponding texture files for them are also downloaded from the same website. A program for reading 3DS file was

downloaded from website GAME TUTORIAL (<http://www.gametutorials.com>). We use this program to read 3DS file and get the vertex data. They are rendered by drawing triangles formed by the related vertexes and applied the correspond texture pieces to those triangles.

5.3.3 Room and ornament picture implementation

Room includes the walls, floor, roof and some ornament pictures on the wall. We defined a base class Side. The room class has 6 side objects, 4 walls, 1 floor and 1 roof. Ornament picture class has also 6 Side objects, it has a front side which displays the picture, a back side show nothing and 4 flank side in order to make the ornament has a 3D visual effect.

5.4 Film scenes animation implementation

Up until this point, we have used a camera class module and various objects to animate a conversation. By combining all these together, we can animate room conversation between two persons. In this project, we implement three film scenes: one scene for two people having a sit-down conversation; one stand while the other sitting; and one scene using differently implemented methods to show a conversation in which one person sites while the other walks around. For convenience of reading, we will use **Red** to refer to the red person and **Blue** to refer to the blue person.

According to the analysis of chapter 2, in a dialogue between two people, a filmmaker might begin with an apex view of both actors, and then alternate views of each. To implement this scene, we initialize the camera at a position such that its **y** and **z** coordinate are in the middle of the **y** and **z** coordinates of the two persons, and its **x** is at one end of the table. From that position, the camera looks at **Red**. View evolution is implemented by making three shots change smoothly. For the first shot, the camera looks at **Red** across the shoulder of **Blue**. According to film grammar, the person we are looking at is occupies roughly 2/3 of the screen while the other one occupies roughly 1/3 of the screen. For the second shot, the camera moves gradually to make **Red** occupy roughly 1/3 of the screen and face the camera while **Blue** person is still back to camera but occupying 2/3 of the screen. In the third shot, the camera moves smoothly to look at **Blue**, giving him roughly 2/3 of the screen. The pictures below show part of the three related shots.



Figure 5.6 A shot - Camera looks at Red

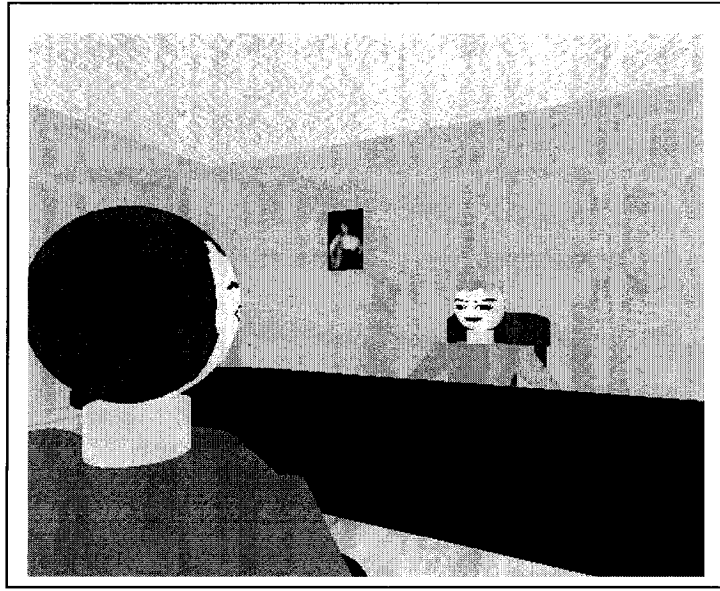


Figure 5.7 A shot - Camera looks at Blue

To film a person approaching another person, a three-shot idiom has to be completed. Let's call the red person **Red**, and the blue person **Blue**. The first shot is a close-up; the **Red** begins in the center of the screen and exits left. The second shot begins with a long view of the **Blue**; the **Red** enters from off-screen right, and the shot ends when **Red** reaches the center. The final shot begins with a medium view of **Blue**, with **Red** entering from off-screen right and stopping at center. We implemented this by having the camera look at **Red** before he begins to walk. Then the camera looks at **Blue**, and keeps still until **Red** approaches the blue person and enters the viewpoint of camera. Figure 5.8 shows one such shot.

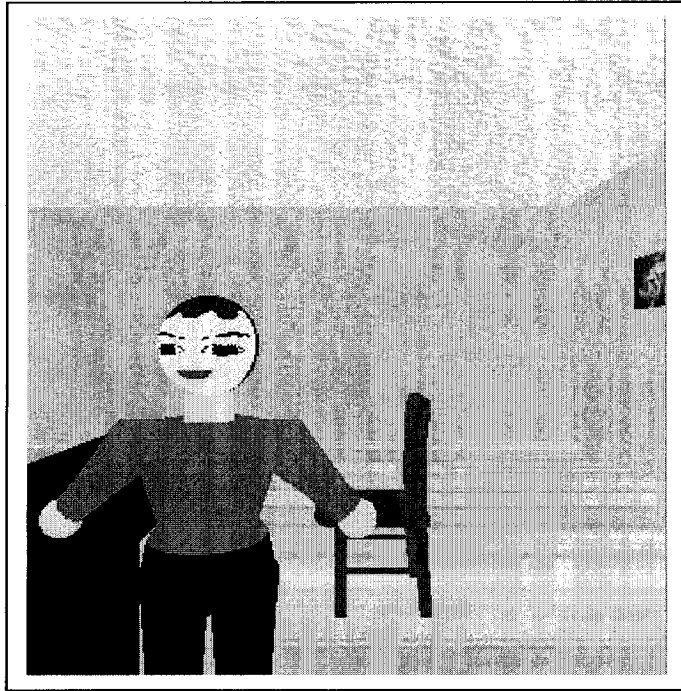


Figure 5.8 Shot 1 of three-shots idiom.
Camera look at **Red** and gives a close-up shot.

Figure 5.9 shows another four-shot idiom that depicts the short-range motion of one actor approaching another. First, The **Red** walks in a straight path toward the camera site. Then the camera pans to follow him. Then the moving **Red** began his walk with his back to the camera and concluded by arriving at a profile position near **Blue**.

To implement the above scene, first, we make camera looks at **Red** when he begins to walk. Next the camera pans to look at the **Red**'s side, then the camera looks at his back and ends by Red arriving at a profile position near **Blue** . The pictures below shows two of these shots.

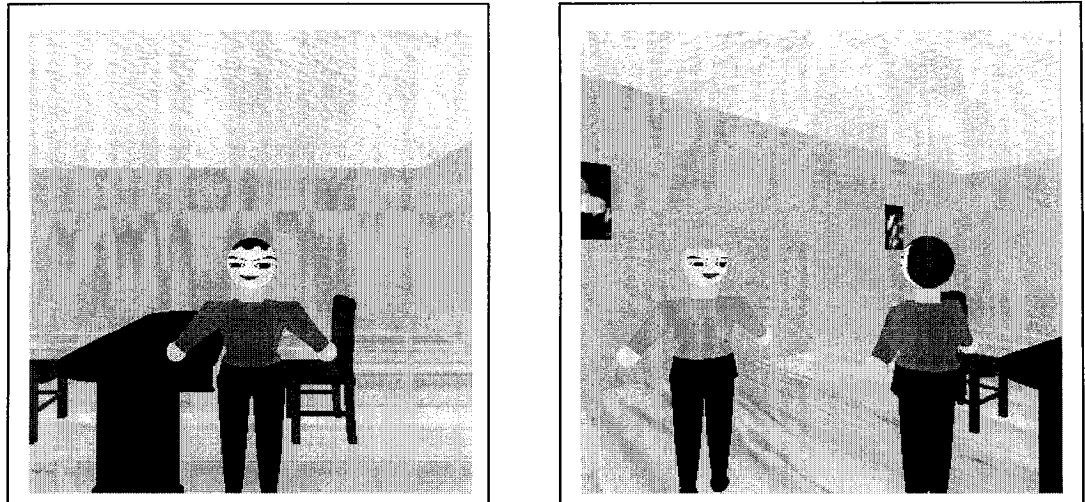


Figure 5.9 Two film shots of four-shots idiom. Shot 1 and shot 3.

To film two people when one is standing up while the other is sitting down, we make the two people perform different actions and calculate the camera position to animate the viewpoint when the sitting person looks at the standing person, also the standing person looks at the sitting person. Two shots are animated here. Figure 5.10 below show these two situations.

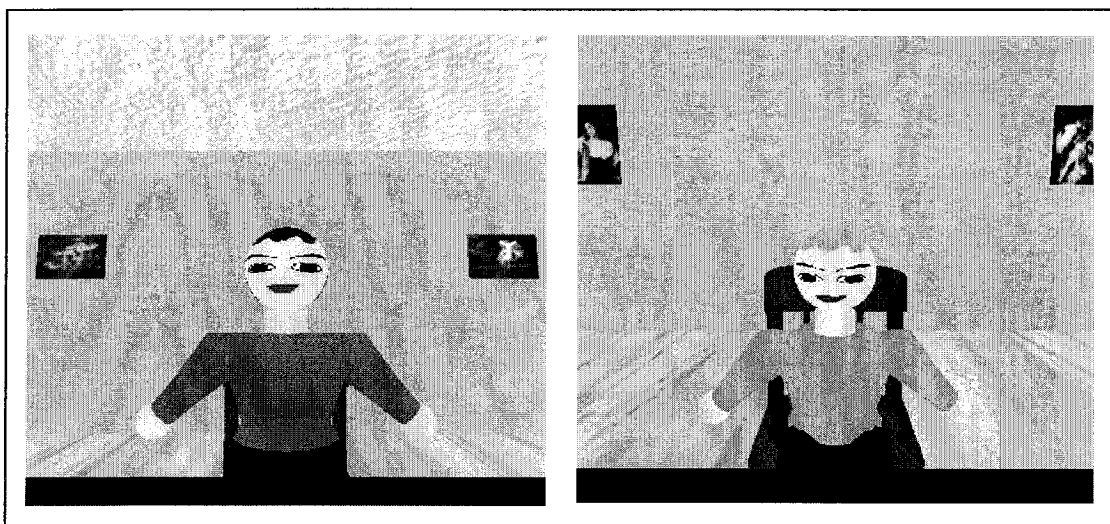


Figure 5.10 Two shots for two players. One is standing while another is sitting

Chapter 6 Results

The results of the camera position calculating algorithms and the project are shown in figure 6.1 through 6.30. The computer used for this project is an AMD Athlon XP2400+, 1.79Ghz, 256MB memory. Figure 6.1-6.17 are animated film grammar scenes for two person conversation and **Red** approaches **Blue**, while other pictures are showed to demonstrate the camera position algorithm, which always places the observed people at the correct position on the screen according to film grammar.

6.1 Film scenes animation results

6.1.1 Conversation when both people sit down

In the first shot, the camera looks at **Red** across the shoulder of **Blue**; according to film grammar, the person observed occupies roughly $2/3$ of the screen while the other person occupies roughly $1/3$ of the screen. In the second shot, the camera moves gradually to make **Red** occupy roughly $1/3$ of the screen and face the camera, while **Blue** still has her back to the camera but occupies $2/3$ of the screen. In the third shot, the camera cuts away to look at **Blue** and place her roughly at the $2/3$ line on the screen. Table 1 below shows the parameters regarding calculating camera positions in scene 1.

Scene Description	Figure No.	Parameters			
		Coefficient k	Angle θ	Person Width M	Screen Width L
Scene 1:					
Shot1 Facing Red	Figure 6.1	3.5	90°	2	3
Shot2 Facing Red, while camera begin to switch to Blue .	Figure 6.2	3.5	90°	1	3
Shot3 Facing Blue	Figure 6.3	3.5	90°	2	3
Shot4 Facing Blue , while camera begin to switch to Red .	Figure 6.4	3.5	90°	1	3

Table 1. Parameters for camera position calculating in scene 1

The pictures below show the four related shots.



Figure 6.1 Scene 1 shot 1 – facing Red

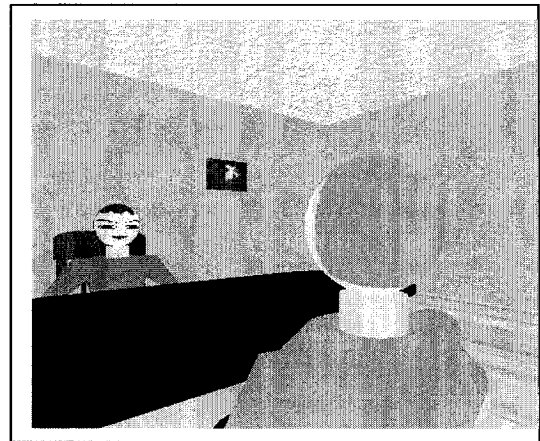


Figure 6.2 Scene 1 shot 2 – facing Red while camera begin to switch to **Blue**



Figure 6.3 Scene 1 shot 3– look at Blue

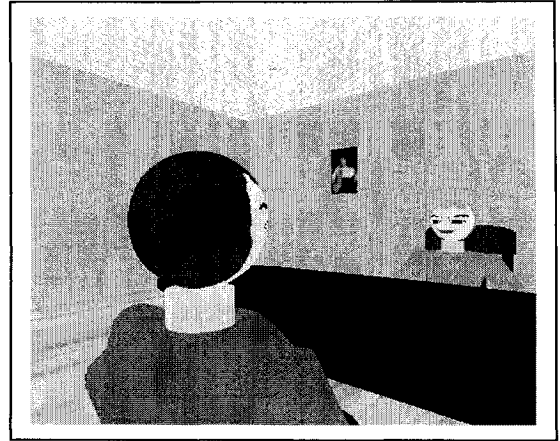


Figure 6.4 Scene 1 shot 4 – look at Blue while camera begin to switch to **Red**

6.1.2 Conversation when one sits down, another stands up

To film two person when one is standing up while another sits down, besides the camera can be placed at a place that is similar to that in a both players sit down conversation, it can also be placed at a position that makes the scene looks as if the sitting person looks at the standing person, or as if the standing person looks at the sitting person.

So the animated shots are listed below. Figures below show these situations.



Figure 6.5 Scene 2 shot 1, facing Red

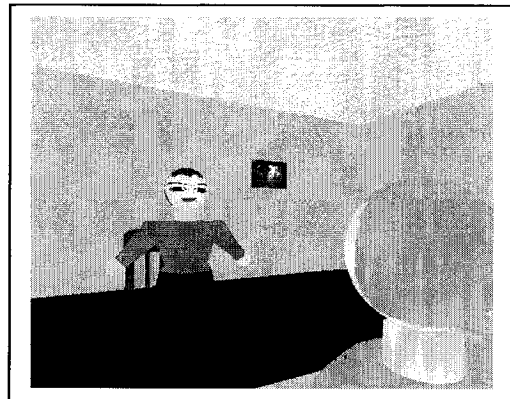


Figure 6.6 Scene 2 shot 2 facing Red

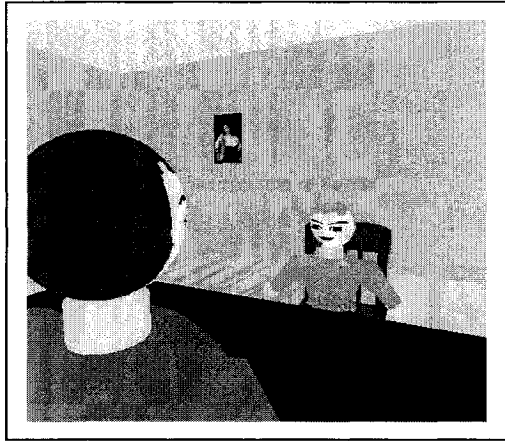


Figure 6.7 Scene 2 shot 3, facing Blue

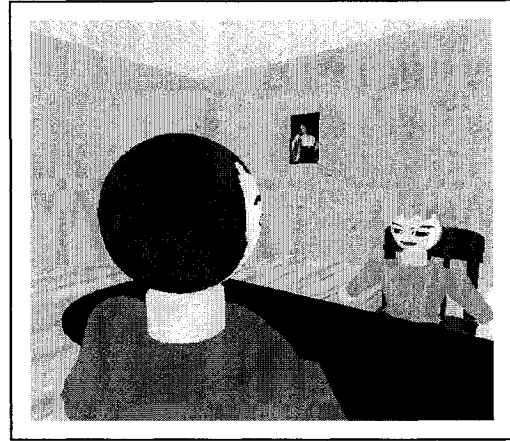


Figure 6.8 Scene 2 shot 4, facing Blue



Figure 6.9 Scene 2 shot 5, look at Red
(An internal reverse shot)

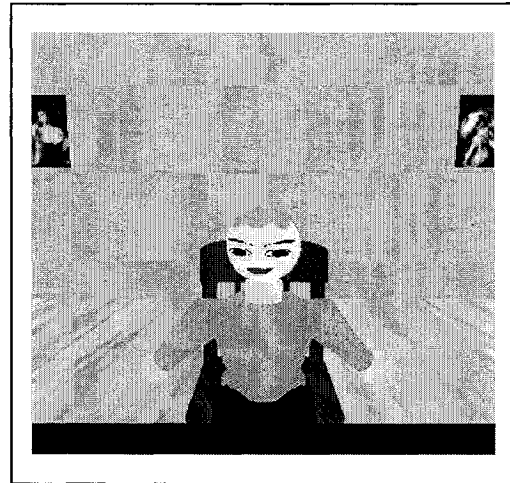


Figure 6.10 Scene 2 shot 6, look at
Blue


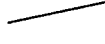
Scene Description	Figure No.	Parameters			
		Coefficient k	Angle θ	Person Width M	Screen Width L
Scene 2:					
Shot1 Facing Red	Figure 6.5	3.5	90°	2	3
Shot2 Facing Red , while camera begin to switch to Blue .	Figure 6.6	3.5	90°	1	3
Shot3 Facing Blue	Figure 6.7	3.5	90°	2	3
Shot4 Facing Blue , while camera begin to switch to Red .	Figure 6.8	3.5	90°	1	3
Shot5 Internal reversal, look at Red	Figure 6.9	3.5		3	3
Shot6 Internal reversal, look at Blue	Figure 6.10	3.5		3	3

Table 2. Parameters for camera position calculating in scene 2

6.1.3 Red moves toward Blue

6.1.3.1 Three-shot animation

A three-shot idioms has to be completed. In the first shot, **Red** begins in the center of the screen and exits left. The second shot begins with a long view of **Blue**; **Red** enters from off-screen right, and the shots ends when **Red** reaches the center.

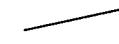
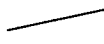
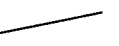
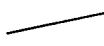
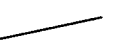
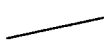
Scene Description	Figure No.	Parameters			
		Coefficient k	Angle θ	Person Width M	Screen Width L
Scene 3:					
Shot1 Look at Red , and Red goes away from screen	Figure 6.11			2	3
Shot2 Red enters into screen from right	Figure 6.12			1	2
Shot3 Camera moves forward closer to both	Figure 6.13			1	2

Table 3. Parameters for camera position calculating in scene 3

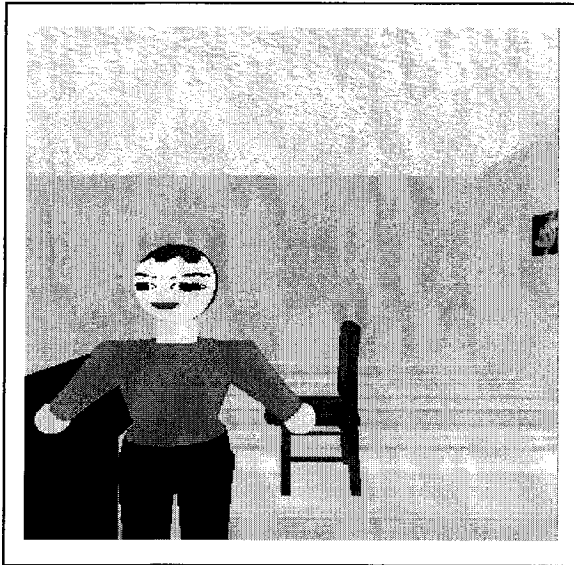


Figure 6.11 Scene 3 shot 1. Look at Red, and Red goes away from screen

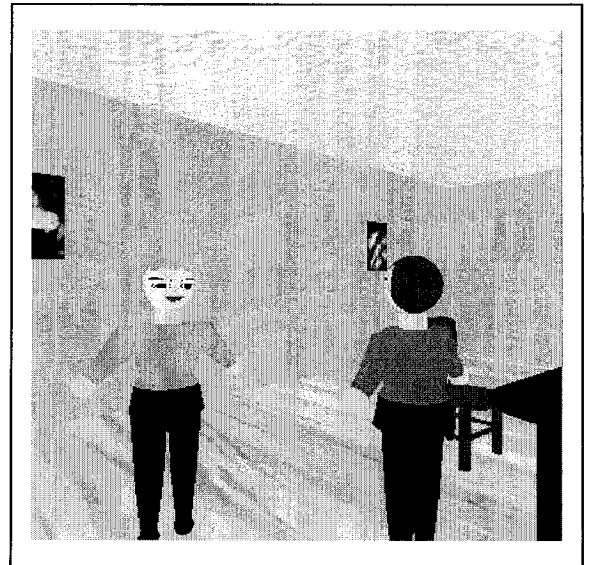


Figure 6.12 Scene 3 shot 2. Shot2 Red enters into screen from right

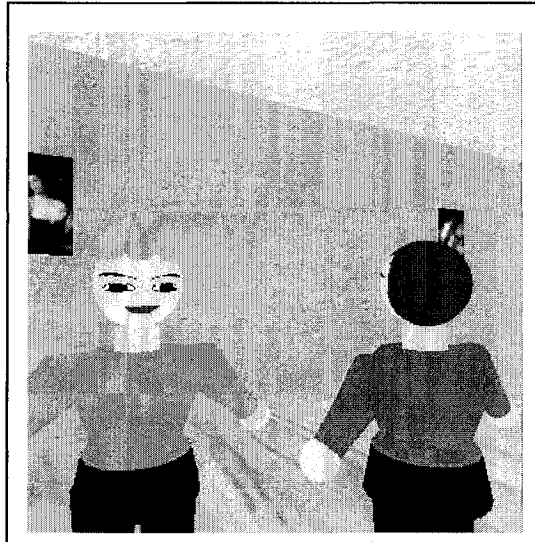


Figure 6.13 Scene 3 shot 3. Camera moves forward closer to both

6.1.3.2 Four-shot animation

Figures below shows another four-shot idiom to depict short range motion of one actor approaching another. **Red** walks in a straight path toward the camera site. Then the camera pans to follow the person to show the 180° pan used in the first shot. Then the moving **Red** began her walk with his back to the camera and concluded by arriving at a profile position. The pictures below shows these shots.

Scene Description	Figure No.	Parameters			
		Coefficient k	Angle θ	Person Width M	Screen Width L
Scene 4:					
Shot 1 Looks at Red	Figure 6.14			3	3
Shot 2 Camera pan to look at Red	Figure 6.15			3	3
Shot 3 Red walks with his back to camera	Figure 6.16			1	2
Shot 4 Camera moves forward closer to both	Figure 6.17			1	2

Table 4. Parameters for camera position calculating in scene 3

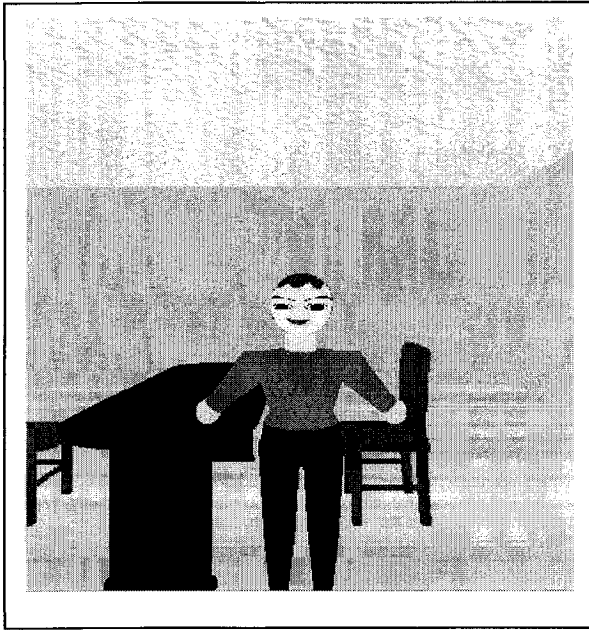


Figure 6.14 Scene 4, shot 1

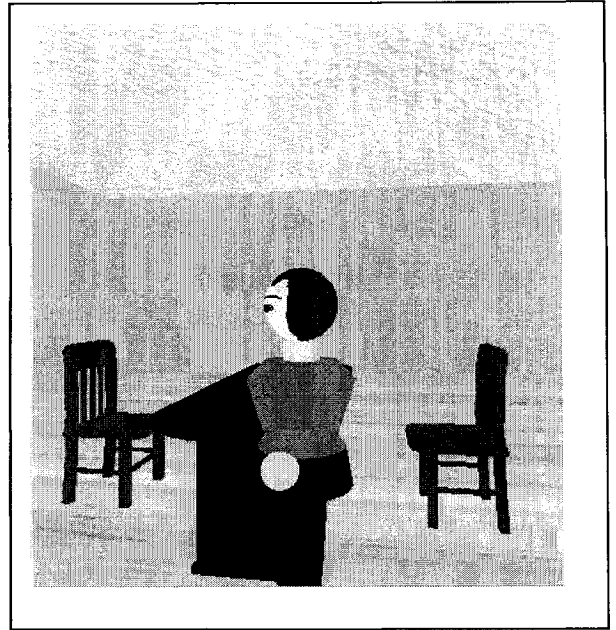


Figure 6.15 Scene 4, shot 2

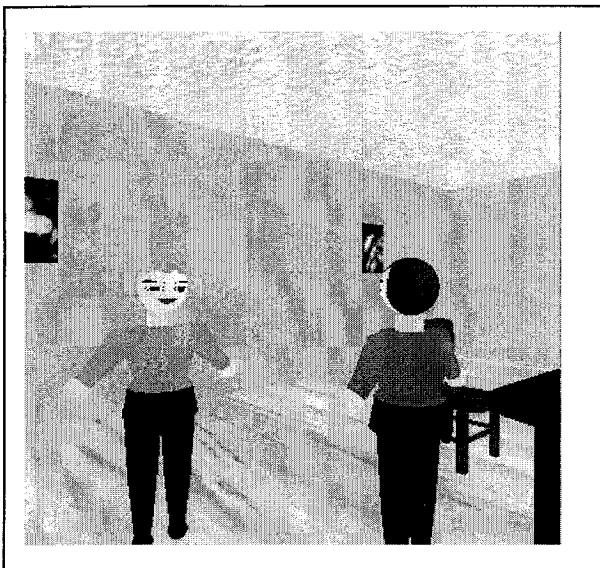


Figure 6.16 Scene 4 shot 3

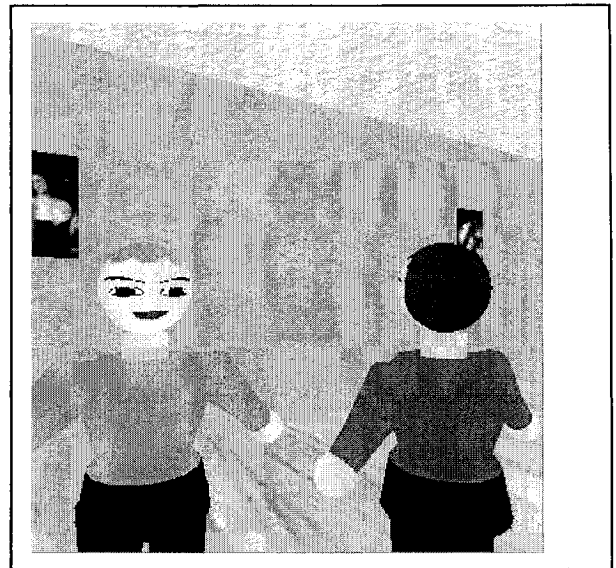


Figure 6.17 Scene 4 shot 4

6.2 Cinematic camera position algorithm results

Figure 6.18-6.30 show the results of camera position algorithm. From the figures, we know that, using this algorithm, we can always figure out a camera position to keep one

person displayed at a specified line M/L of the screen, while the person facing away displayed at $(L-M)/L$ line of the screen (for example, $L=3, M=2$).

Shot Description	Figure No.	Parameters			
		Coefficient k	Angle θ	Person Width M	Screen Width L
Shot 1. Facing Red . Red stands up and begins to walk	Figure 6.18	1.28	90°	2	3
Shot 2. Facing Red . Red walks to right corner of table opposite to Blue .	Figure 6.19	2.7	270°	2	3
Shot 3. Facing Red . Red walks to right corner of table.	Figure 6.20	1.7	264°	2	3
Shot 4. Facing Red . Red passes the line of Blue	Figure 6.21	2.3	60°	2	3
Shot 5. Facing Red . Red passes Blue from his back	Figure 6.22	2.1	81°	2	3
Shot 6. Facing Red . Red passes Blue from his back. Red occupies 1/3 of screen	Figure 6.23	2.1	81°	1	3
Shot 7. Facing Red . Red walks to the left of Blue on his side.	Figure 6.24	1.0	210°	2	3
Shot 8. Facing Red . Red passes Blue from his left.	Figure 6.25	0.7	266°	1	3
Shot 9. Facing Red . Red passes Blue from his left to the corner of table opposite to Blue	Figure 6.26	0.4	196°	2	3
Shot 10. Facing Red . Red turns left at the corner of table opposite to Blue	Figure 6.27	1.75	275°	2	3
Shot 11. Facing Red . An internal reverse shot when Red begins to walk to the table corner opposite to Blue	Figure 6.28	1.05	84.5	2	3
Shot 12. Facing Red . An internal reverse shot when Red turns right at the corner opposite to Blue	Figure 6.29	3.5	90	2	3
Shot 13. A apex position shot	Fig 6.30	1	90	1	2

Table 5. Parameters for camera position calculating some separate shots

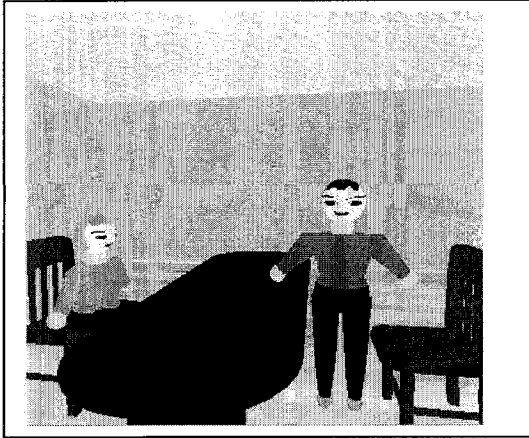


Fig. 6.18 Shot 1. $k = 1.28$, angle = 90

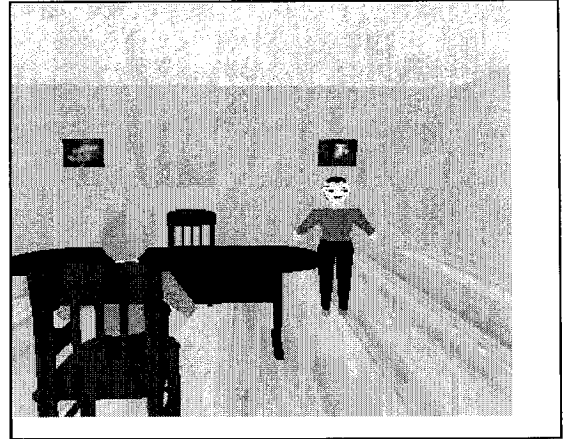


Fig. 6.19 Shot 2 $k=2.7$, angle = 270

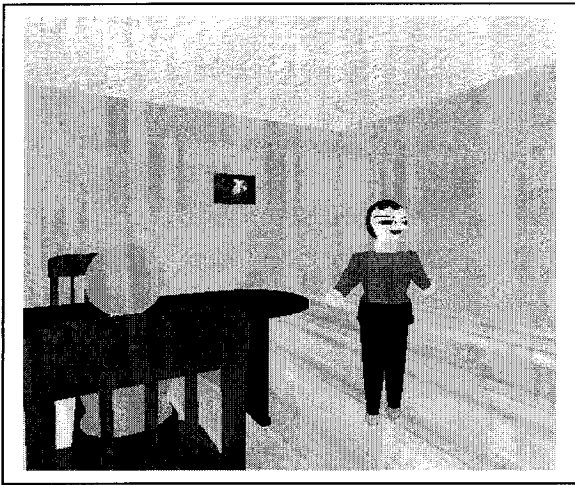


Fig. 6.20 Shot 3. $k=1.7$, angle = 264

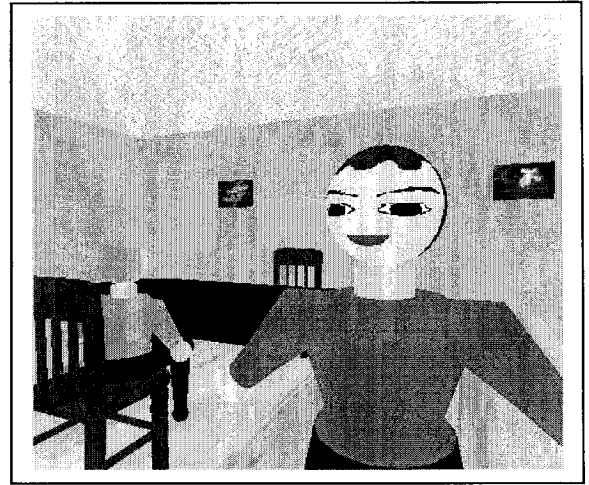


Fig. 6.21 Shot 4. $k=2.3$, angle = 60

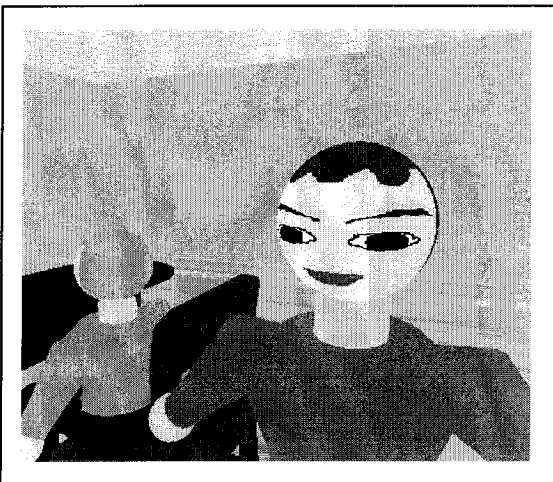


Fig. 6.22 Shot 5. $k=2.1$, angle = 81



Fig. 6.23 Shot 6. $k=2.1$, angle = 81

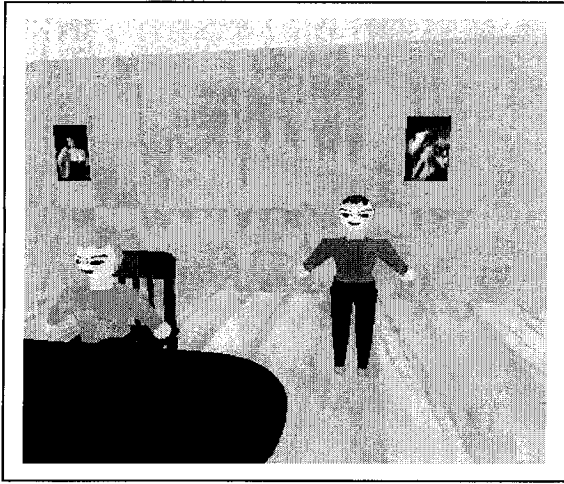


Fig. 6.24 Shot 7. $k=1.0$, angle = 210

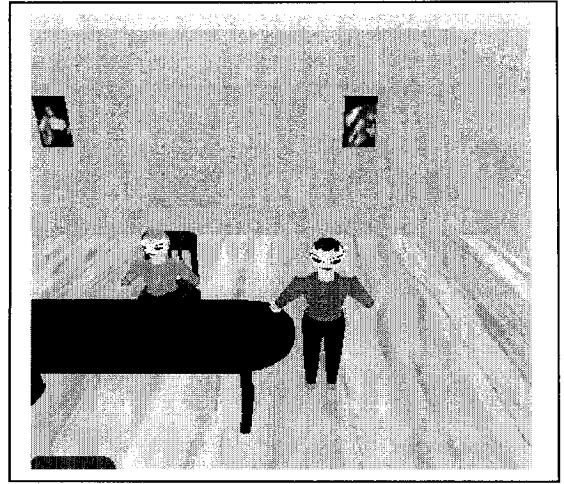


Fig. 6.25 Shot 8. $k = 0.7$, angle = 266

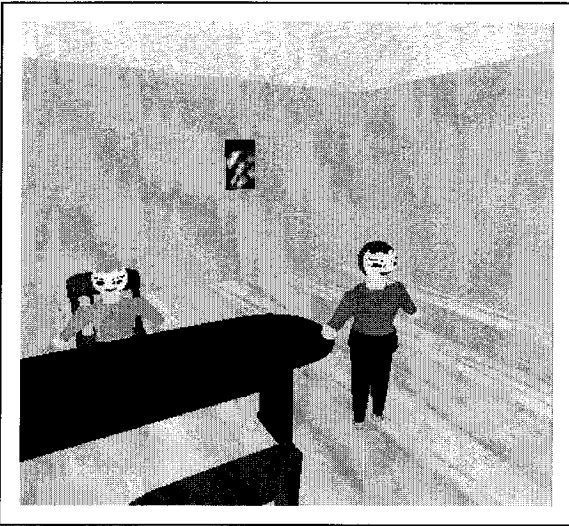


Fig. 6.26 Shot 9. $k = 0.4$, angle = 196

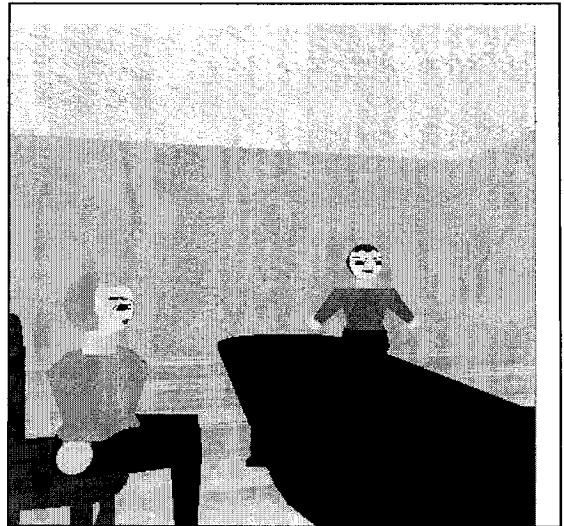


Fig. 6.27 Shot 10. $k = 1.75$, angle = 275

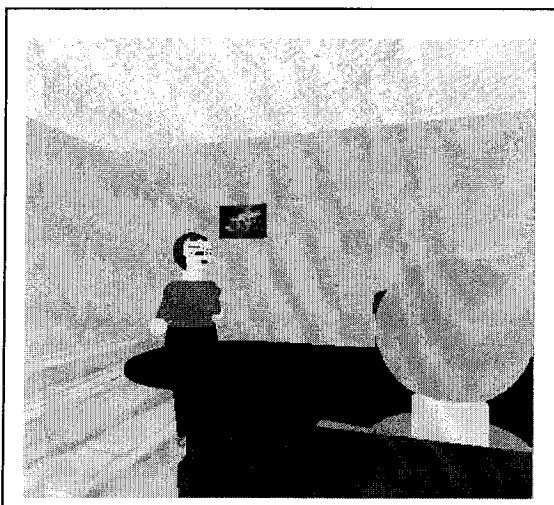


Fig. 6.28 Shot 11. $k = 1.05$, angle = 84.5



Fig. 6.29 Shot 12. $k=3.5$ angle = 90

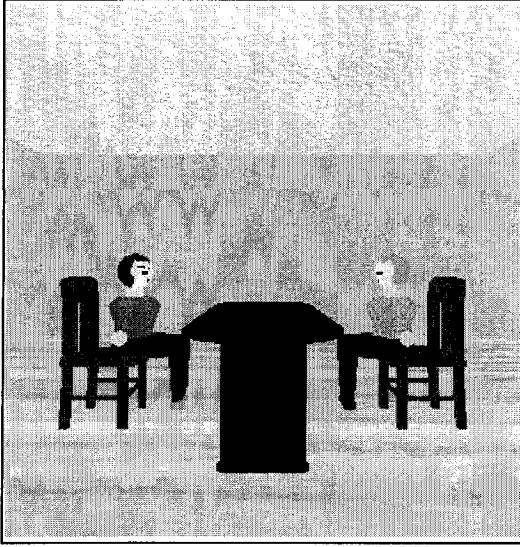


Fig. 6.30 Shot 13. $k=1.0$ angle = 90

Chapter 7 Conclusion and Future Work

7.1 Conclusions

The film industry has plenty of experiences in **camera** techniques and has already developed many heuristic methods and principles. How to apply film grammar in computer graphics **camera control** is a big new research area. In this thesis we have developed algorithms for cinematic camera position calculating which can present the viewers or players good cinematography conversation shots between two players. We have successfully combined trigonometric function and 3D space geometry transformation to solve the problem, and made an effective use of computer graphics and animation techniques to demonstrate our solution. We make the claim that our algorithm is novel and has specific advantages because of the following characteristics:

- According to our research, we have not found an existing algorithm to solve the camera position problem in a conversation. Some papers propose an idea to put camera in a cinematic way, but have not given an definite algorithm to address the placement of the camera.
- The solution idea is totally creative. Traditionally, when we do coordinate transform from 3D world coordinate to 3D view transform, eventually to 2D coordinate system. We would build equations related with model view matrix, projection matrix and viewport transformation matrix, and we will find lots of

square root in these functions. And it would be very hard to solve those equations, and may not even could solve it because could not get enough equations to get the desired variables (e.g., x , y , z value of camera and x , y , z value of model, total 6 variables here).

- When we build related functions, although we could not have enough equations to solve those 6 variables (camera and model coordinates). We made some 3D spatial transformation to make the equations simple and easy to solve. Of course, users have to input some parameters to get an unique value for the 6 variable, but that's reasonable since there are lots of possible values for those variables. When users give the parameters, it will give the correspond value for each variable.
- It can compute the camera and model position automatically according to the position of two people in a conversation and some user input parameters. And it adapts to many situations in a two players conversation, for example, *external* reverse and *internal* reverse, *right angle* and *apex* position, etc., and can always give a proper camera and model position and give the desired cinematic visual effect.
- The algorithms we provide in this thesis is useful for game program developer or film makers to do story boarding. By using a combination of c++ and OpenGL we have minimized the amount of code that a programmer has to write.

7.2 Future work

Our original goal of work in this thesis is to provide an algorithm for cinematic conversation camera control and apply it in some conversation situations. While the solution described has successfully accomplished the initial goal, it also has some problem that should be addressed in future work.

- **Parameter adjustment.** Although the camera can arrange the observed person in the desired position on the screen, it is straightforward to all users to adjust the parameters needed to get a good camera position.
- **Not always give a good visual effect.** Although the camera can arrange the two players in the desired position on the screen, but sometimes couldn't give a good visual effect.
- **Get an camera position when there are more than two players in a shot.** For graphics animation, two people conversation is only one situation of film shots. There could be three or more people in a shot. Future research in this direction will consider these conditions and give a good camera position to arrange all the players in the desired positions and give a good visual effect according to film grammar.

References

[Arijon 76] D. Arijon. *Grammar of the Film Language*. New York : Communication Arts Books, Hastings House, publishers, 1976.

[Katz 91] Steven D. Katz. *Film directing shot by shot*. Studio City, CA., Michael Wiese Productions, 1991, P121-156.

[Drucker 95] Steven M. Drucker and D. Zeltzer, *CamDroid: A system for Implementing Intelligent Camera Control*. In *Proceedings of the SIGGRAPH Symposium on Interactive 3D Graphics '95*, 1995

[Christianson 96] David B. Christianson, Sean E. Anderson, Li-wei He, *Declarative Camera Control for Automatic Cinematography*, In *Proceedings of the AAAI-96*, August 1996.

[Lin 04] Lin, Ting-Chieh, Zen-Chung Shih, Yu-Ting Tsai, *Cinematic Camera Control in 3D Computer Games*. *WSCG SHORT Communication papers proceedings WSCG'2004*, February 2-6, 2004, Plzen, Czech Republic. Available on line from http://wscg.zcu.cz/wscg2004/Papers_2004_Short/G31.pdf

[Drucker 92] Steven M. Drucker, Tinsley A. Galyean, D. Zeltzer, *CINEMA: A system for procedural Camera Movements*. *Proceedings of the 1992 symposium on Interactive 3D*

graphics. Cambridge, Massachusetts, United States Pages: 67 – 70, 1992

[Bares 97] William H. Bares, James C. Lester, *Cinematographic User Models for Automated Realtime Camera Control in Dynamic 3D Environments, Proceedings of the Sixth International Conference, UM97*. Vienna, New York: Springer Wien New York. © CISM, 1997. Available on-line from <http://um.org>.

[Amerson 01] D. Amerson and S. Kime. *Real-time cinematic camera control for interactive narratives*, In The Working Notes of the AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment, Stanford, CA, March 2001.

[Grogono 02] Peter. Grogono, *COMP 6761 Advanced Computer Graphics, Lecture Notes*, 2002

[Grogono 98] Peter. Grogono, *Getting Started with OpenGL, Supplementary Course Notes for COMP 471 and COMP 6761*, 1998

[Hearn 03] Donald Hearn, M. Pauline Baker, *Computer Graphics with OpenGL*, Prentice Hall, 2003

[Woo 99] Mason Woo, Jackie Neider, and Tom Davis, *OpenGL: Programming Guide, Third Edition, The Official Guide to Learning OpenGL, Version 1.2*, Addison Wesley Developers 1999

Web based references

1. Concordia University Graphics Library: <http://www.cs.concordia.ca/~grogono/CUGL/>
2. Graphics Example programs: <http://www.cs.concordia.ca/~grogono/Graphics/graphex.html>
3. OpenGL, the industry standard for 2D & 3D graphics:
<http://www.opengl.org/about/overview.html#1>
4. NeHe productions: <http://nehe.gamedev.net/>
5. 3D CAFE: <http://www.3dcafe.com/asp/househld.asp>
6. Game Tutorial: <http://www.gametutorials.com/gtstore/c-1-test-cat.aspx>
7. GLUT User Interface Library: <http://www.nigels.com/glt/glui/>