Virtual Cinematography for Computer Games

James Kneafsey
M.Sc. in Computing

Institute of Technology
Blanchardstown

2006

Supervisor: Hugh McCabe

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Masters in Computer Science in the Institute of Technology Blanchardstown, is entirely my own work except where otherwise stated, and has not been submitted for assessment for an academic purpose at this or any other institution other than in partial fulfilment of the requirements of that stated above.

Signed: _____ Dated: ____/____/____

Abstract

Modern computer games employ various styles and approaches to controlling the player's view of the game. They all rely on the use of a virtual camera which shares many of the characteristics of a real-world motion picture camera and the operations of these are guided by cinematography. Cinematography represents over a century of the evolution of camera work that provides visual variety, varying subjectivity and a generally more engaging experience for the viewer. So far, computer game developers have only applied cinematography to their work in a limited fashion. We propose that more aspects of cinematography can be adapted for use in a vast number of game genres than have been up to now. In this thesis we examine the work that has been conducted thus far in the area of automatic camera control in a 3D virtual environment and present our virtual cinematography system for 3D shooter games. This system was developed and tested with the Quake II game engine. In preparation for its development, we researched principles and guidelines from cinematography and formalised those that we consider to be relevant to 3D shooter games into algorithms. We also provide an approach to room detection in a virtual environment as part of this virtual cinematography system.

Acknowledgements

I would firstly like to thank my supervisor Hugh McCabe for his overall guidance and instruction throughout my master's. Also, I wish to acknowledge the support and motivation I received from my family and friends, in particular Christine, which helped me through the various stages. Finally, I wish to thank my fellow postgrads for their assistance in solving numerous problems along the way.

Contents

Figures

Chapter 1

Introduction

1.1 Background

Our work concerns the depiction of action in interactive 3D virtual environments, namely those viewed on a visual display unit where interaction is generally facilitated by use of a combination of a computer keyboard and mouse, or a game controller. Within this area we focus on 3D computer games, in particular, first-person shooter (FPS) games. These games require the user to engage in combat with computer-controlled characters and possibly other players. The player's view of the game world is provided by a virtual camera but this is employed in a relatively simple manner with the camera adopting the viewpoint of the virtual character that the player controls which is called the *avatar*. We believe this can be improved upon. Sophisticated camera work can be seen in *cut-scenes*. These are non-interactive, pre-animated, pre-rendered scenes that are inserted between sections of normal game-play and display superior character and camera movements. However, we are concerned only with the camera work during game-play itself. We have turned to cinematography as a guide for raising the level of sophistication of camera work within an FPS game.

Cinematography (Brown 2002; Mascelli 1965) describes principles and techniques pertaining to the effective use of cameras to film live action. The correct application of these principles and techniques produces filmed content that is more engaging, compelling and absorbing for the viewer. The virtual cameras employed by 3D computer games are used to provide the player with an appropriate view of the game world. These can simulate all of the functionality of their real-world counterparts yet little effort is

usually made to incorporate cinematographic techniques and principles into their operation. The aim of our work is to apply these techniques and principles to the virtual camera in a FPS game.

1.2 Cameras in Computer Graphics

The use of virtual cameras has a long history in computer graphics (Foley et al. 1990). To examine the techniques used in this field it is useful to consider those used to record images in the real world. Images are recorded on a real world camera by the projection of light from 3D objects onto a 2D *image plane*. The recording mechanism which coincides with this plane is traditionally magnetic film. The portion of the 3D world that is projected onto the image plane and, therefore, that is seen by the camera, is bounded by a volume in space positioned at the image plane called the *pyramid of vision* (Figure 1.1).

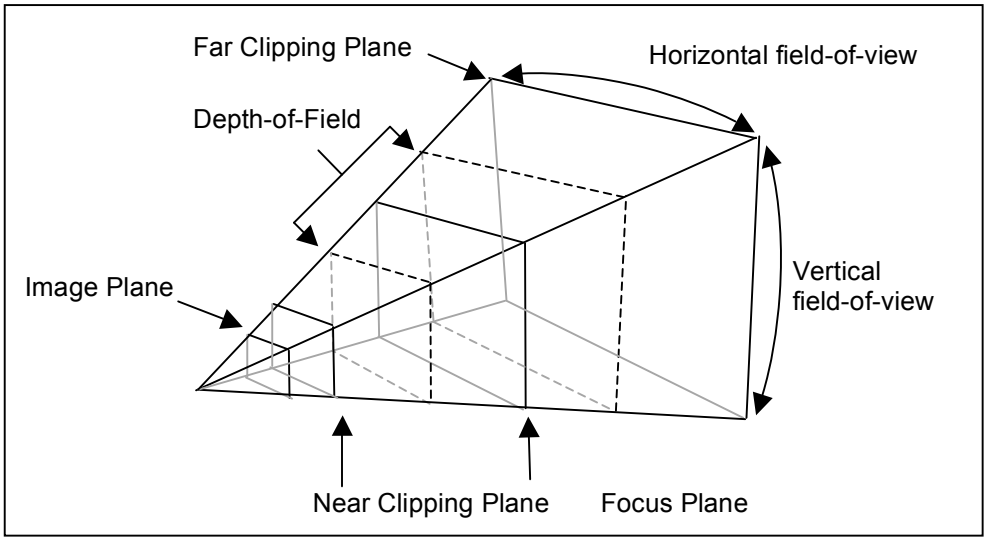

Figure 1.1: The pyramid of vision or viewing frustum

The parameters that define the pyramid of vision are the near and far clipping planes and the horizontal and vertical fields-of-view. The near and far clipping planes are planes perpendicular to the line-of-sight in front of which and behind which, respectively, nothing is visible. The horizontal and vertical fields-of-view are the angles around the vertical and horizontal axes, respectively, of the camera through which objects are visible. The portion of space that is visible takes the shape of a truncated pyramid or

frustum. This *viewing frustum* is ultimately determined by the focal length of the lens of a real world camera which in turn is determined by the distance from the lens to the image plane and by the curvature of the lens. The focal length also controls the focal plane, another plane that is perpendicular to the line-of-sight at which objects are in sharp focus. Two more planes represent the edges of the depth-of-field. Objects within the depth-of-field are in focus; those outside it are blurred.

Mathematically, a virtual camera records images in the same manner (Foley et al. 1990). Projections are made in the 3D world from objects within the viewing frustum onto the image plane, in this case simply a rectangular area at a designated position and orientation in space. The virtual camera is capable of generating images from any viewpoint within the 3D scene and of mimicking the different focal lengths and fields-of-view of real world cameras. Unlike their real world counterparts, virtual cameras are not subject to any physical constraints and can be moved around the scene in any way we want. This affords rich possibilities for animated and interactive content. Virtual cameras also differ from real ones in that objects in a virtual world are resolved into a sharp image by default so if a sense of a depth-of-field is required, i.e. that some objects are out of focus, additional calculations must be performed.

Early computer games did not utilise virtual cameras of the type described above. They either represented a 2D world, which rendered the virtual camera unnecessary, or else used a scrolling isometric view, giving only the appearance of a 3D world (DeMaria and Wilson 2002). First-person shooter (FPS) games like Wolfenstein 3D and DOOM were among the first to give the player a moving viewpoint in a 3D world and granted the player full freedom to move around this world. These and many other types of games require the player to control the avatar in order to carry out tasks required by the game's rules. With FPS games in particular, the player adopts her avatar's view of the virtual environment hence the name first-person shooter (first-person perspective will be explained in Chapter 2). This is an example of a simple, functional view used in a game that we believe can be improved upon with cinematography. It is the id Software game Quake II that we have chosen to modify for the purposes of this project.

1.3 Cinematography

We now outline some fundamental ideas of cinematography in order to present it as a suitable basis for camera control in a 3D computer game. Cinematography formalises numerous aspects of film-making such that good practises can be identified and expanded upon. Obviously our focus is on guidelines that pertain to the operation of cameras. At the lowest level cameras are used to capture *shots*. A shot is the smallest unit of filming; it is the filmed content resulting from the time when the camera is turned on until it is turned off. A number of shots filmed in the same setting combine to form a *scene* and a number of scenes, in turn, form a *sequence*. Different camera angles can be used during a shot: *objective camera angles* and *subjective camera angles*. This quality depends on the proximity and movement of the camera relative to the action. There are a number of different types of shots that may be used, each one presenting its subject at a different size on the screen. The choice of *shot type* determines the proximity of the camera to the action and therefore the subjectivity level of the shot. The point at which one shot ends and the next one begins is known as a *cut* and the timing of cuts is a key consideration. There are also directives to apply to shots which require a moving camera so that the viewer is granted a consistent view in terms of the positions and movements of the elements on the screen. Cinematography also specifies how to combine shots into scenes and ensure consistency throughout.

Computer games can be analysed in the terms presented above. Generally a level in a game consists of one shot. If we consider the example of FPS games, a continuous view of each level is granted without cuts. However, multiple shots are present in one specific area of games: cut-scenes. As mentioned, cut-scenes are non-interactive sections of the game where impressive visuals are used in terms of the character models, character animations and the camera work. The camera is animated “by hand” giving its human operator complete freedom to execute impressive movements and cuts. Although cut-scenes are one of the areas where the most impressive camera work has been applied in terms of cinematography, we do not use this approach because of the need for human control of the camera and the outcome of identical camera work on every viewing of a given cut-scene. Our work focuses on camera work in the interactive part of the game.

1.4 Aims and Objectives

Our general aim was to develop a novel camera system for FPS games that incorporates cinematographic techniques and principles into its operation. In particular this system should be capable of:

- Providing more relevant views of the game content that are not limited to the player's point of view
- Varying the level of subjectivity throughout the game
- Adding dramatic emphasis where necessary

If successful, the system should do the above in such a way that:

- It is consistent with cinematography practice
- It has minimal impact on the performance of the player
- It enhances the game-play experience for the player

In addition to this, some practical objectives were to:

- Implement the system in such a way that it can be easily incorporated into any relevant game engine
- Implement the system such that it can be easily extended with new features.

1.5 Achievements

In this thesis we describe in detail our work and present our results. The achievements of the work can be summarised as follows:

- Analysis of relevant work that attempts to incorporate cinematographic principles into a virtual camera's operation in a virtual environment and of principles and techniques in cinematography texts
- Application of a number of these principles and techniques to the camera system in a popular FPS game, id Software's Quake II, in accordance with the aims and

objectives outlined above, specifically, such that the system is extensible and portable

- Documentation of camera-specific portions of the Quake II game engine
- Introduction of the notion of `CameraBots` as a means of filming action within the game from multiple angles
- Design of a finite state machine to model the action in the game, control editing decisions and camera repositionings, and so on
- Design of a room detection algorithm for 3D virtual environments
- Testing of this system by a number of participants and gathering of results from this test with respect to its impact on each player's performance and game-play experience

1.6 Thesis Structure

This introduction has outlined the background of our research namely that of computer graphics, computer games and cinematography. It has presented our reasons for choosing cinematography as a source of camera control principles, FPS games as an area of application and has outlined our aims, objectives and achievements.

In Chapter 2 we discuss the aspects of cinematography, theoretical and practical, that we consider suitable for application to FPS games. Chapter 3 covers previous work concerning camera control in a 3D virtual environment and how our own work builds upon it. In Chapter 4 we outline the various components of our own work and illustrate their basis in cinematography. In Chapter 5 we discuss the work we performed on the Quake II computer game and specific algorithms. Finally, in Chapter 6 we present results from tests perform on our system, consider possible future work, and provide some conclusions.

Chapter 2

Cinematography

In this chapter we introduce the principles, terminology and guidelines from cinematography that we believe are suitable for application to camera operations in a 3D computer game. We have formalised a number of these guidelines and principles and converted them into algorithms for our virtual cinematography system. For a fuller treatment of the following we direct the reader to classic texts such as Mascelli (1965) and Browne (2002).

2.1 Cinematography Fundamentals

Cinematography is defined as the art of film making (Browne 2002). It has evolved as an art form over a period of more than a century and represents a language for presenting live action in different ways. More specifically it describes a set of guidelines and principles for the effective use of lighting, positioning of characters and props, and operation of cameras in order to film. The successful application of these guidelines and principles results in a high standard of filming and a coherent and engaging depiction of the subject matter. The placement of the camera can take otherwise inert content and dramatise it through visual manipulation by determining the objects that are visible, the location of the camera relative to the objects and the perception of depth. By these means it is possible to add dramatic emphasis where required and ultimately to evoke an emotional response in the viewer. It is also possible to communicate supplementary information regarding the events to the viewer with the use of specific camera angles. Throughout this chapter some of the principal roles with regard to the filming of a motion picture are referred to, so we provide broad definitions here. On a film set, it is the

director who chooses the events that are to be filmed, the *cinematographer* who coordinates the camera operators to film the events subject to certain principles from cinematography and the *editor* who edits the resulting *footage* (the filmed content) to produce a coherent depiction of the events.

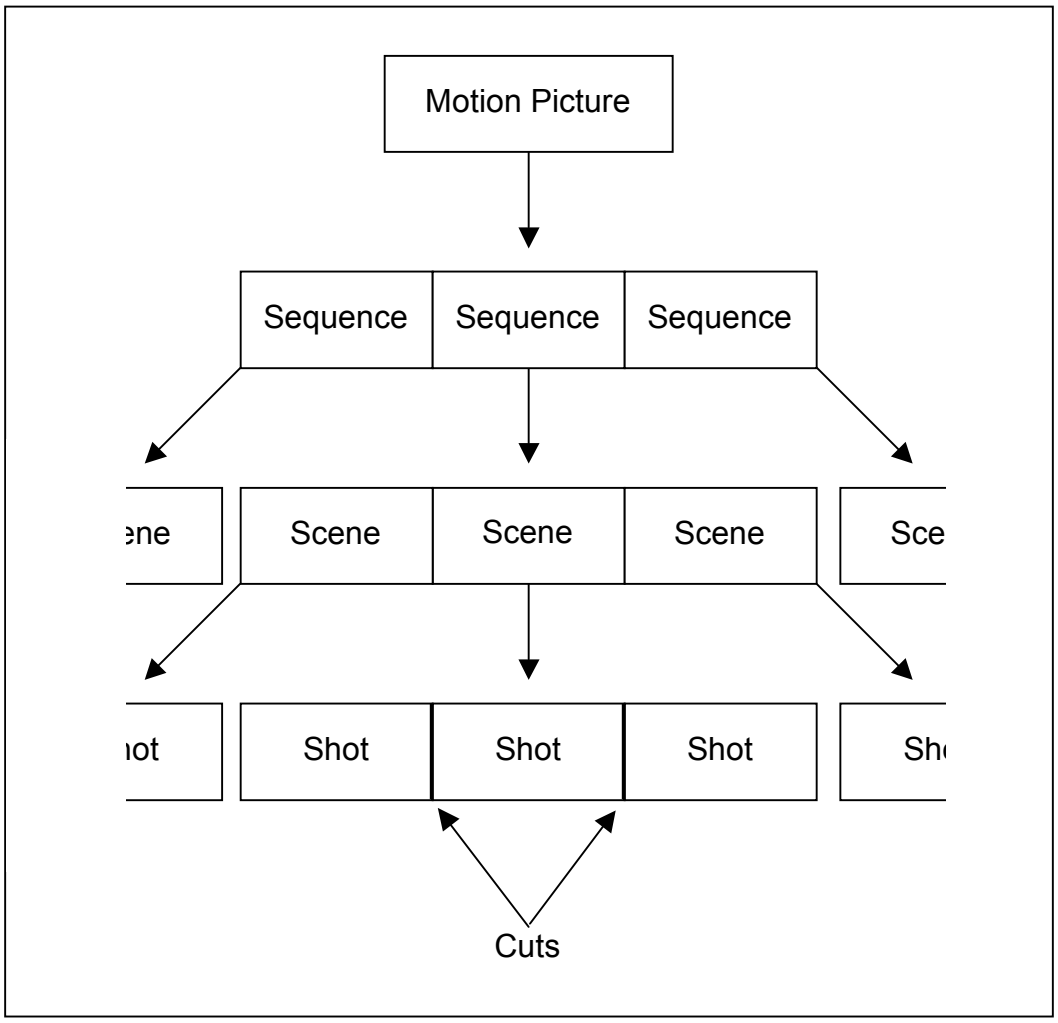

Figure 2.1: The breakdown of a motion picture

We now introduce some basic cinematography definitions and concepts which will be elaborated upon over the remainder of the chapter. At the lowest level a motion picture is composed of *shots* (Mascelli 1965), a shot being a continuous view filmed by one camera without interruption (Figure 2.1). The type of camera angle used in a shot determines how much the camera, and therefore the viewer, engages with the action depicted in the shot. Camera angles that are closer to and move with the action are more *subjective*;

angles that are more detached and seem to move independently of the action are more *objective*. Each shot also has a composition, i.e. the arrangement of actors and props on the screen, where the *frame* might be balanced, in terms of the objects within it, in different ways. The frame is the boundary outside which nothing is recorded by the camera; the parts of the scene that are visible are said to be in the frame. During a shot the camera may be stationary or it may employ simple or complex movements depending on the content of the scene and the mood that is to be established. Cinematography provides guidelines on how to position and move the camera. *Cinematic continuity* provides for the prevention of viewer disorientation and a consistent presentation of events.

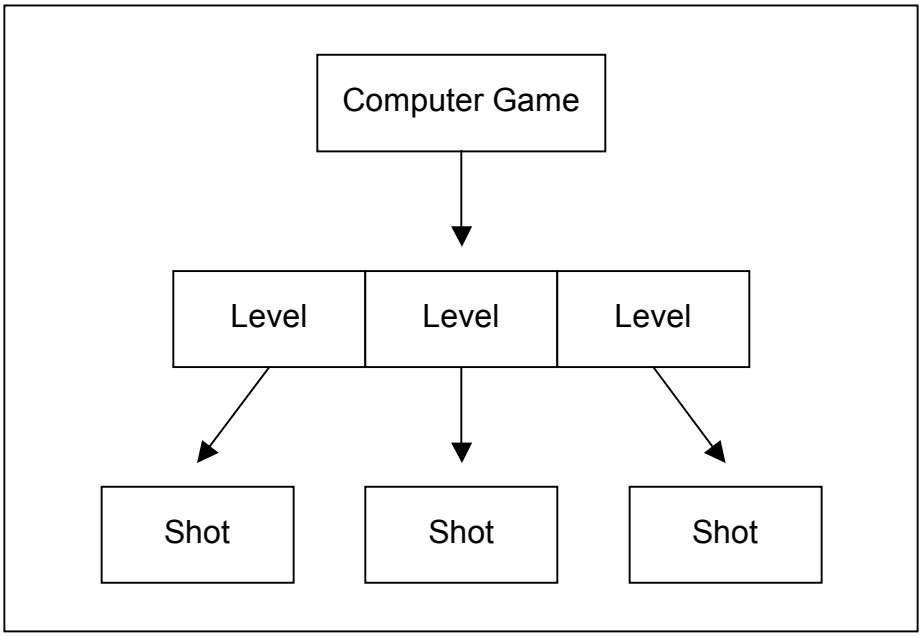

Figure 2.2: The breakdown of a computer game

The instantaneous transition between two shots is known as a *cut*. Although cuts are carried out by the editor after a motion picture is filmed, they must be performed in real time for an interactive virtual cinematography system. There are a number of guidelines regarding the timing of a cut in the depiction of live action. The combination of a number of shots makes up a *scene* which is a single setting where action takes place during a particular time period. There are different approaches to filming scenes. Some of these require the use of single cameras and others multiple cameras; some allow for multiple

takes, i.e. filming the same shot multiple times until the director is satisfied, while others must be filmed in a single take. Scenes, in turn, are combined into *sequences.* A number of sequences make up a visual representation of a continuous narrative with a number of different camera placements that may not have necessarily been filmed in a continuous way.

If a similar analysis of computer games is considered (Figure 2.2) it is noted that, in terms of cinematography, they are significantly simpler than films. Typically, computer games consist of a number of levels with each one containing a single shot. There are no cuts during a level. From this point of view, as we explain in Chapter 4, our work converts these long uninterrupted shots into a series of shots with cuts generated in real time.

2.2 Shots

The shot is the unit of viewing. It is the footage recorded by a single camera from the time it is turned on until it is turned off. The combination of a number of different types of shots produces a scene. When filming a shot the following factors all have to be considered and a decision made in each case (Brown 2002):

- *Camera angle*: The type of camera angle used determines the shot's level of subjectivity and therefore whether the viewer becomes drawn into the story or remains detached.
- *Shot type*: The type of shot taken decides how much of the scene and the characters in it are framed.
- *Lens height*: The height of the lens relative to the characters influences the viewer's relationship with the characters.
- *Subject angle:* The angle the lens is placed at relative to the characters communicates more or less of a sense of depth of the scene.
- *Lens type:* Different types of lenses can be used to determine the portion of the scene that is focussed as well as the perception of depth.

We now describe these different factors in more detail.

2.2.1 Types of Camera Angles

The type of camera angle used in a shot determines its dramatic impact. A more su*bjective* camera angle draws the viewer into the scene and transfers the emotions the characters are feeling onto the viewer, but may also disorient the viewer if used for too long. A more o*bjective* angle grants a general view of the scene. For this the camera is distant from the action and films a large area of the scene such that it provides the view of a neutral observer often called a *third-person perspective*. Thus viewers are less likely to empathise with the characters and the characters seem unaware of the camera's presence. An objective camera angle can help the viewer find his bearings within a scene and so may be used at intervals for this purpose.

A subjective camera angle gives the viewer a feeling of being in the scene. Thus he is more likely to empathise with the characters in the scene but, on the other hand, may become disoriented if this type of angle is overused. This angle is achieved by placing the camera closer to the action and moving it along with the action, for example alongside a speeding car or in-between combatants fighting in a battle. There is no dividing line between objectivity and subjectivity in a camera placement but rather a gradual transition from one to the other.

A particularly subjective camera treatment is the *first-person perspective* (Figure 2.3[1]). This is when the camera provides the view of one of the characters in the scene. In order to switch to a first-person perspective successfully it is usually important to clarify to the viewer that he is seeing what the character sees, perhaps with a preceding shot of the character looking into the distance. Generally when using this kind of view, care must be taken because other characters interacting with the character, whose viewpoint the

[1] Computer game characters and scenes are used throughout this chapter to illustrate features of cinematography. Most of the images (this one being an exception) are from modified versions of a game or game development environments and so the features being illustrated do not already exist in the game, but are being used for purposes of illustration.

camera has adopted, may look directly into the lens which may startle the viewer and make them aware of the camera. If this is an issue, a *point of view* angle may be used instead. To film this angle the camera is positioned beside a character rather than simulating her view. This means that when other characters look at her, they do not look directly into the lens but to the side of the lens. Now the viewer feels as though he is standing beside the character rather than seeing her view. A point of view angle is as close as the camera can get to a fully subjective treatment while maintaining some objectivity.

Figure 2.3: The first-person perspective. The hand of the character whose view is being shown is visible in the foreground

2.2.2 Types of Shots

Different types of shots have different functions as a scene and characters are introduced and the narrative develops (Brown 2002). Shots are defined in relation to the proportion of the subject in the frame. This is determined by the distance from the subject to the lens and the type of lens used. The longer the lens, the larger the subject will be in the frame (see Section 2.2.5).

Scenes are often opened with establishing shots (see Section 2.7) to give the viewer a general view. An *extreme long shot* is an example of an *establishing shot*. It portrays a vast landscape, cityscape or similar area. Often a *wide angle lens* (see Section 2.2.5) or a

pan (a rotating movement of the camera around its vertical axis) is used to take in such a great expanse.

Figure 2.4: Long shot

Another establishing shot, the *long shot* (Figure 2.4) is used to establish the geography on a smaller scale. It includes the characters and the setting, for example, a room or a street and sets out the locations of the characters before the camera is moved in closer. The camera will return to an establishing shot, often a long shot, after filming a number of closer shots. This allows the viewer to become reacquainted with the layout of the setting.

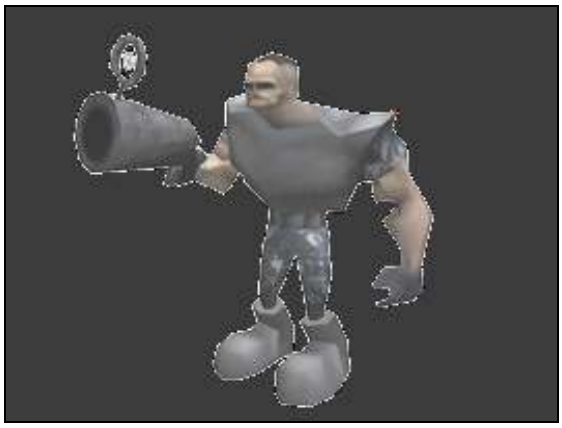

Figure 2.5: Full shot

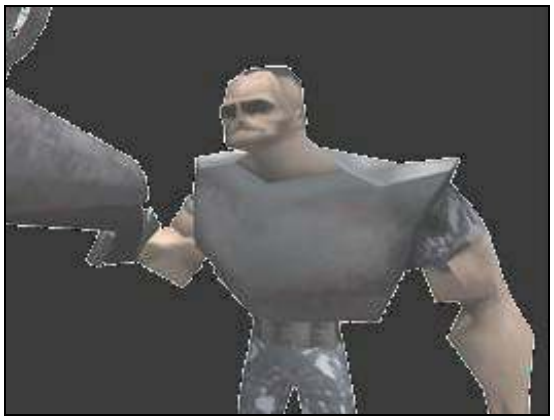

Figure 2.6: Medium shot

Character shots bring the audience closer to the characters so it is possible to make out specific actions. The *full shot* and *medium shot* (Figure 2.5 and Figure 2.6 respectively) are both used to frame one or a number of characters. A full shot depicts the entire subject and a medium shot frames the subject from just above the knees or just below the waist up.

Figure 2.7: Medium close-up

Figure 2.8: Head and shoulders close-up

Figure 2.9: Head close-up

Figure 2.10: Choker

Figure 2.11: Tight close-up

Close-ups are used to isolate a portion of the scene, to add dramatic emphasis to a part of the narrative and to portray a character's facial expressions. There are a number of different types of *close-ups* (Figure 2.7 to Figure 2.11):

- A *medium close-up* frames the subject from between the waist and shoulders to the top of the head.
- A *head and shoulders close-up* is from below the shoulders to above the head.
- A *head close-up* frames just the head.
- A *choker* is from just below the lips to just above the eyes.
- A *tight close-up* is similar to a choker but with less of the chin and forehead.
- An *extreme close-up* can be used to frame just an actor's eyes or a small object such as a coin.

The terminology for character shots differs according to the number of characters in the shot. A *single* is any close-up, medium shot or full shot with a single character. A *clean single* doesn't include any other characters. A *dirty single* includes a small part of another character closer to the camera than the main subject. *Two-shots* and *three-shots* are shots of two and three characters respectively.

A commonly used type of shot is the *over-the-shoulder* shot. This is a medium shot or close-up of one character taken over the shoulder of another. It is often used in dialogue sequences with two characters opposing each other. The camera may begin with a two-

shot, move in for alternating over-the-shoulder shots of each character and then shoot point of view close-ups. Over-the-shoulder shots should be filmed such that the cheek line but no facial features of the foreground character are seen. She should be framed such that either, the head and part of the shoulder, or just the side of the head is visible.

Figure 2.12: Cut-in: The shot on the right represents a cut-in from the shot on the left

Some shots have a specific function with respect to the preceding shot. These include the *cut-in*, a shot that magnifies a section of the screen to highlight some action while continuing the main narrative (Figure 2.12). A *cut-away* is a shot of something off-screen such as a reaction shot, i.e. a shot of a character reacting to an event in the previous shot. It can also be used for clarification, for example to film something off-screen that a character was looking at in the preceding shot. Similar to a cut-in, an *insert* highlights a portion of the previous shot but it differs in that it presents this new shot from a different angle and/or focal length. Inserts are used to add dramatic emphasis to an event or to provide additional information related to the narrative.

A *reverse shot* (Figure 2.13) depicts the scene from the opposite direction to the previous shot. For example a character walks towards and past the camera in one shot and in the next shot the camera is turned around so that the character walks away from the camera. Care must be taken when using a reverse shot because it may transpose the actors so that they appear on opposite sides of the screen to the previous shot. This may temporarily confuse the viewer (see Section 2.5).

Figure 2.13: Reverse shot: The character is exiting screen right in the shot on the left and the camera rotates to film him entering screen left in the shot on the right

2.2.3 Lens Height

The height of the lens relative to the subject can be varied to add psychological and dramatic overtones to the narrative and influence the relationship between the viewer and the characters on the screen.

Figure 2.14: Level angle

Eye-level shots are filmed with a *level angle* lens (Figure 2.14). Objective level angle shots are filmed from the height of a person of average stature. Point of view level angle shots are filmed from the height of the person whose point of view the lens adopts.

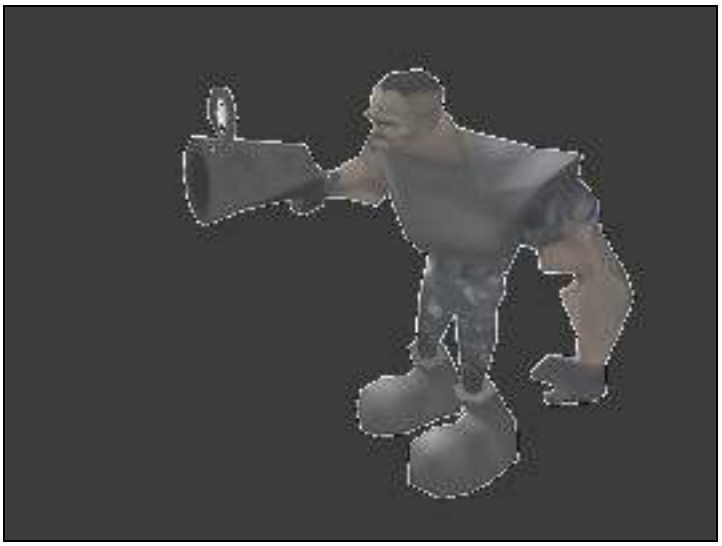

Figure 2.15: High angle

High angle shots (Figure 2.15) are achieved by positioning the camera above the subject and angling it down towards the subject. These shots are used to establish a setting containing vast geography or to film action in depth. They also can make the audience feel superior to the subject or simply to make the character seem smaller.

Figure 2.16: Low angle

The camera is positioned below the subject and angled upwards for a *low angle* shot (Figure 2.16). These are employed to increase the apparent height or speed of the subject

and also for shots of a subject that is dominating or who holds a position of authority, possibly from the point of view of another character.

2.2.4 Subject Angle

Figure 2.17: Three-quarter angle

Figure 2.18: Profile

When filming a scene it is usually desirable to communicate a sense of depth and the three-dimensionality of the scene to the viewer. This is achieved by filming from an angle that allows more sides of the subject to be seen. Therefore subjects filmed from an angle such as a three-quarter or 45º angle (Figure 2.17) seem more three-dimensional than those pictured in profile, for example (Figure 2.18), because the front and side of the subject are visible. *Angle-plus-angle* means using a high or low angle in addition to filming the front and side of the subject, perhaps with a three-quarter angle. The audience, therefore, sees the greatest number of sides of the subject granting a greater sense of the three-dimensionality.

Figure 2.19: Dutch tilt

A natural disaster such as an earthquake or the feelings of an intoxicated or drugged character can be illustrated on-screen with a *Dutch tilt* (Figure 2.19). A Dutch tilt is implemented by placing the vertical axis of the camera at an angle to the vertical axis of the setting. The camera must be tilted by a sufficient amount to make the tilt look intentional rather than accidental.

2.2.5 Types of Lenses

The type of lens used further determines the area that is filmed and the depth of that area that is in focus. *Wide angle lenses* (Figure 2.20) take in more of the visual field than normal human vision (Figure 2.21). They exaggerate linear perspective so that distant objects appear smaller than usual, very near objects appear so large that their shape is distorted and objects appear more distant from each other than normally. This expansion of space has the effect of exaggerating the movements of objects towards and away from the camera. *Long angle lenses* (Figure 2.22) reduce the effect of linear perspective so that distant objects do not appear as small as usual and objects appear to be closer together. This compression of space means that moving objects appear to move slower.

Figure 2.20: Wide angle lens

Figure 2.21: Field-of-view approximate to human vision (same viewpoint)

Figure 2.22: Long angle lens

Depth-of-field is the proportion of the visible scene that is in focus and is dependent on the aperture of the lens being used and the distance between the lens and the subject. In most situations a wide angle lens will offer a greater depth-of-field. A reduced depth-of-field does have its uses: A long angle lens can be used to keep only a small area of the scene in focus to highlight a particular event or character, perhaps to isolate the protagonist in a crowd. Mascelli (1965) suggests that the camera should focus on either the character that is most prominent in the frame or the person experiencing the greatest emotion. Sometimes it may be necessary to shift focus from a character or object in the foreground to an object farther away. This is called *rack focus*. As already mentioned, 3D

computer games render all objects in focus so depth-of-field must actually be added to rendering algorithms if required.

2.2.6 Using Shots

Brown (1965) and Mascelli (2002) suggest that shots be employed according to the following principles. Each shot should only depict the characters and the area of the scene that are essential to the story at that point. A scene may open with a long shot or an extreme long shot to establish the setting (more about this in Section 2.7). A medium shot introduces the group of characters. Close-ups isolate a particular character or emphasise an event. Long shots re-establish characters' location in relation to the background and facilitate their motion about the scene. For important interaction between two characters a medium or a two-shot is used. Extreme close-ups depict very small objects. The camera progresses inwards as the action develops and moves back out to re-establish the setting or to portray new developments. When the setting is established, medium, medium close and close shots are used.

The optimal angle to use for a close-up is a three-quarter angle. This gives the subject a more rounded appearance and depicts more of his facial features. A profile angle, on the other hand, makes his face look flat and doesn't have the same eye-to-eye relationship between the audience and actor. A straight-on point of view close-up can also be effective if three-quarter lighting is used to give the face three-dimensionality. Objective close-ups should usually be filmed from the height of the subject. If previous shots use a high or low angle, however, the close-up should use a similar angle. Subjective close-ups should also be filmed from the subject's height so the audience will relate more with him; it is not as effective to have the audience looking up or down at him. Point of view close-ups should be shot from the height of the actor whose point of view the camera takes. Over-the-shoulder close-ups should be shot from a level as close to the height of the actor facing the camera as possible while still framing the foreground actor appropriately.

A series of close-ups should use similar image sizes and camera angles. For example, a series of back-and-forth shots of opposing actors in a two-shot should depict opposing matched angles (i.e. the actors face opposite sides of the screen) and similar image sizes

for objective or point of view shots; a series of reaction shots, of a crowd for example, should also use matched angles and image sizes. One instance where a close-up might not be suitable is in the case of a subject that may move around a lot in the frame. A larger static frame is more appropriate in this case than a smaller frame that has to follow the subject's movements.

We believe the general use of the camera operations and principles with relation to shots as discussed in this section is lacking in the majority of computer games and that using a number of different types of shots with varying subjectivity, lens height, lens type and so on would enrich the gaming experience. Close-ups would add dramatic emphasis to a particular subject's actions or reaction; long shots would capture more complex actions and so on.

2.3 Composition

Composition refers to the arrangement of actors and props on the screen. Its function is to provide order to the objects and to direct the viewer's eye to the most significant elements in the setting. It can also evoke an emotional response in the viewer. The factors contributing to the composition of the screen are *balance* and the *centre of interest.* It is the combination of composition and the movement of subject matter on the screen that holds the viewer's attention. Although most of these principles were not applied to our own work, as obvious difficulties arise when trying to employ them in the context of an interactive system, we present them here for the sake of completeness and as an avenue for future research.

2.3.1 Balance

To evoke the required response from the audience whether that is contentment or frustration it is necessary to consider some contributing factors. Each object on the screen possesses a *pictorial weight* which means that one area of the screen may seem "heavier" than another. More pictorial weight is given by:

- Larger objects

- Moving objects
- Objects closer to the top of the screen than to the bottom
- Objects closer to the right of the screen than to the left
- Isolated objects rather than crowded objects

If the purpose of a given shot is to evoke contentment, the camera should be positioned such that the frame is balanced; if, on the other hand, frustration is the desired effect the factors discussed above should be used to compose an unbalanced frame. In order to compose a balanced frame one might use *formal balance* or *informal balance.* A *formally balanced* frame has a symmetrical composition with neither side of the screen being favoured in terms of pictorial weight. Formal balance suggests calm and lack of energy. The opposite is true of *informal balance*: There is a dominant object on one side of the screen but a contrasting object of equal weight on the other. A large object may reside on one side with a small but moving object on the other. The dominant object should attract more attention perhaps by a higher placement in the frame.

2.3.2 Centre of Interest

The second factor to consider with respect to composition is the centre of interest on the screen. Typically only one centre of interest should be used at any instant during a scene; otherwise a number of elements on the screen compete for the viewer's attention. The centre of interest is the dominant part of the frame. It could be a single object or a group of objects favoured by camera treatment, positioning, movement or other conditions. Exceptional circumstances such as fighting in the scene might warrant the use of more than one centre of interest to represent the chaos in the content.

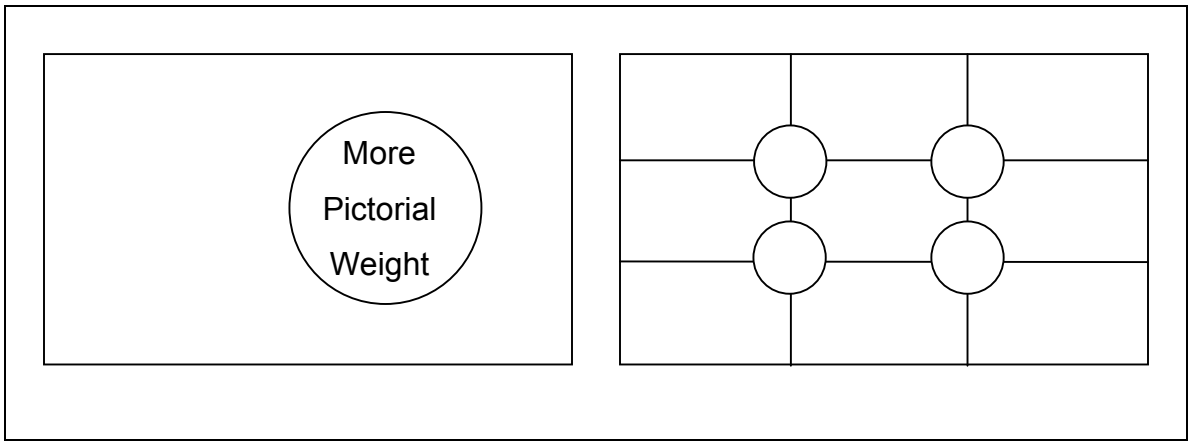

Figure 2.23: Placement of the centre of interest on the screen: It should either be towards the right side of the screen (left image) or at any intersection of the four lines illustrated (right image) as indicated by the circles above

The centre of interest should usually not be placed at the centre of the screen but rather towards the right where there is greater pictorial weight (Figure 2.23). Alternatively, the screen can be divided into three equal parts horizontally and vertically and the centre of interest placed at one of the four intersections of the dividing lines. However the centre of interest should not always be limited to these placements and should be repositioned from shot to shot for visual variety.

The centre of interest can be changed from one object to another by using selective focus or by positioning or movement, for example, by having the character move towards the camera. The new centre of interest should have a similar position to the previous one so that the viewer does not have to spend time looking for it. For example for a sequence of close-ups each character should take up the same amount of space on the screen and their eyes should be at the same level; a close-up following a long shot of the same object should place the object in the same general area of the screen and so on.

2.3.3 Framing

In addition to the positioning of objects with regard to the composition of the frame there are some general guidelines for framing characters and other objects. A given character should be given enough headroom (the space between the top of the head and the frame) such that the picture is not crowded, but not too much to make it bottom-heavy. If she is

looking towards one side of the screen, her *look* should also be given room because it possesses its own pictorial weight. She should not meet any edge of the frame exactly otherwise it might look as though she is leaning or standing on the frame. She should not be framed at the joints, i.e. at the knees or elbows, but rather in-between. Lastly, as per the centre of interest, objects should generally be placed slightly off-centre.

2.4 Camera Movements

Different types of camera movements are often essential to keep the subject in frame. This section outlines the different types of movements used on a film set.

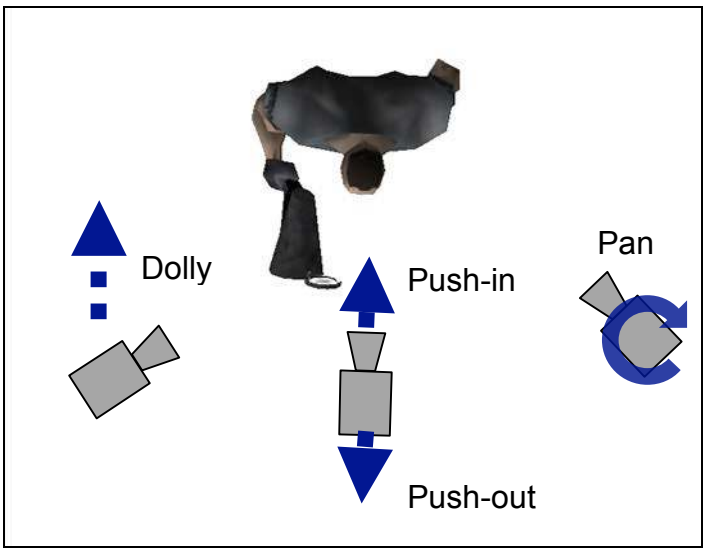

Figure 2.24: Dolly, push-in, push-out and pan

Translational movements of the camera take their names from the apparatus used to carry them out. A *dolly* (Figure 2.24) is any movement of the camera in the horizontal plane. It is named after the wheeled structure that the camera is attached to. To move the camera towards or away from the action specifically is to *push-in* or *push-out* respectively. A *crane*, named after the cranes used on film sets, is any vertical movement of the camera. A combination movement is achieved with a dolly that has a crane attached. It is worth reiterating at this stage that the nature of virtual cameras means they are not subject to physical constraints and, since they do not require this apparatus to operate, can be

moved around the virtual scene with much greater ease and can even achieve shots that would be impossible in the real world.

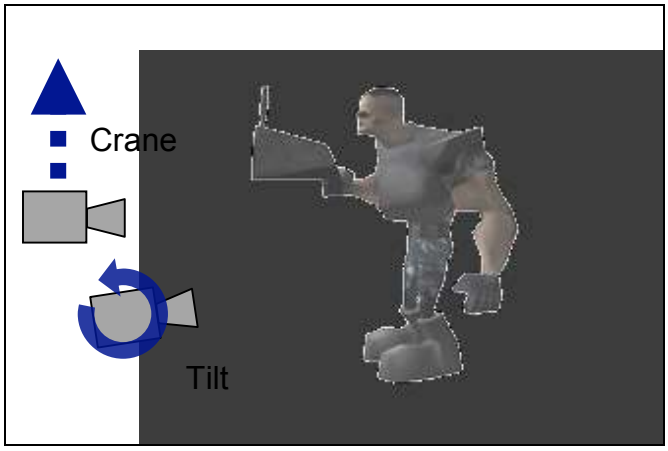

Figure 2.25: Crane and tilt

The camera can also be rotated around its vertical, lateral or longitudinal axes. These movements are known as the *pan, tilt* and *roll* respectively (Figure 2.25). The pan is used to take in the various elements of a scene. The tilt and roll are used in more specialised situations. All can be combined with translational movements for complex camera movements.

Examples of some commonly used camera movements include *tracking,* which is when the camera moves parallel to and facing a character, and *countermove*, for which the camera moves in the opposite direction to its subject while panning to keep the subject in frame (Figure 2.26). The countermove has the effect of making the background move twice as fast as with a tracking shot. A *reveal* is any shot where a significant action, character or object in the scene is kept out of frame until required by the story. A movement of the camera then reveals the action as a development in the narrative. In applying camera moves to the filming of a scene a cinematographer must ensure that each one is motivated and carried out smoothly. The camera should be eased into and out of a move and it should settle at the end of a move before the cut for the next shot.

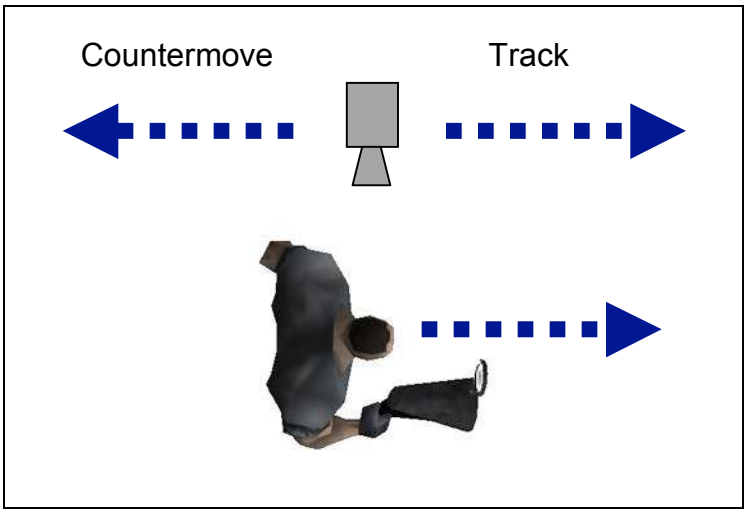

Figure 2.26: Tracking and countermove

2.4.1 Zoom

A *zoom* is used to magnify a portion of the scene by varying its focal length. By zooming in, the lens gradually changes from a wide angle lens to a long angle lens. Thus, the background is compressed and depth-of-field reduced. In some cases, a push-in may be favoured over zooming in because the background moves away as the camera draws closer to the subject giving a greater sense of movement. A *punch-in* is when the lens is switched to a longer angle instantaneously perhaps to provide a close-up of a character in the scene. Again, the background is compressed due to the switch to a longer lens.

2.5 Continuity

Continuity with respect to a motion picture refers to the movements and positions of characters and props, and the looking directions of characters (a character's *look*) on the screen, and whether there is consistency from shot to shot in a given scene (Mascelli 1965). Without continuity the action would quickly become confusing and frustrating to the audience (of course frustrating the audience may be the desired effect in some instances). In the case of computer games, as explained previously, each game level is portrayed with one long continuous shot without cuts so continuity is generally not an issue. If, however, we wish to introduce the concept of numerous shots to a game, it must be taken into account.

2.5.1 Screen Direction and the Action Axis

It is necessary to introduce the concept of *screen direction* to understand how to film the action such that continuity is maintained. Screen direction refers to the direction of movements and positions of characters and props, and the looking directions of characters relative to the screen. More specifically it is whether an object is moving to the right or left of the screen, is positioned on the right or left of the screen, or whether a character is looking towards the right or left of the screen. Continuity means, therefore, that the subject is positioned on, moving towards or looking towards the same side of the screen from shot to shot.

The *action axis* is a means to facilitate consistent screen direction. It is an imaginary vertical plane defined by movement, positioning, looks, specific actions or physical geography within the scene. It comes into play at the end of a shot where another shot of the same scene is to follow. As long as the camera remains within the 180º horizontal arc of one side of the action axis during a cut, it is easy to maintain continuity. The following sections explain this in more detail.

2.5.2 Moving Subjects

Continuity for moving subjects establishes a consistency between movements in the setting and movements on the screen. The action axis defined by a moving subject is a plane along its path of motion at a given instant. If the camera is positioned on one side of the action axis at the end of shot 1, it should be positioned within the 180º arc defined by the action axis at the start of shot 2 (Figure 2.27 below). This means that the subject will be moving towards the same side of the screen at the end of shot 1 and the beginning of shot 2. It is important to clarify that the action axis only comes into play when one shot ends and the next one begins, i.e. it does not matter if the camera crosses it during the shot (thereby exceeding the 180º limitation) because the viewer observes the change.

Another means for the camera to get to the other side of the action axis is to cut to the neutral axis, i.e. by filming along the action axis so that the subject is either moving towards or away from the camera, before cutting to the other side. The whole point is that the direction of motion of the subject on the screen is not instantaneously reversed

without explanation, so if the subject is seen moving in a neutral direction there is no problem. The subject might reverse its own direction within the scene relative to the camera of course, as opposed to its screen direction being altered by a change in camera placement. In this case, the turn should be depicted clearly to avoid confusion.

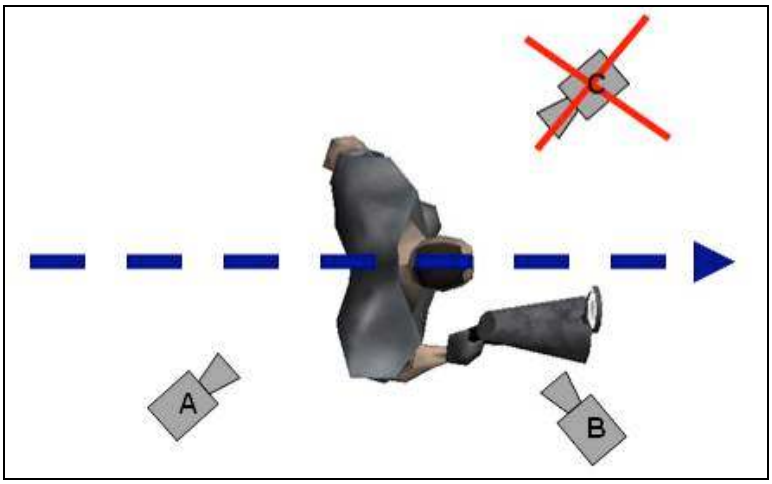

Figure 2.27: The action axis for a moving subject. If the subject is filmed with camera A at the end of shot 1, the camera must keep to this side of the action axis defined by the subject's motion for the beginning of shot 2. Therefore placement B is valid while C is not.

A special case regarding continuity for moving subjects is when the camera is filming a character that walks into another room. There are two schools of thought in this situation. One considers the new room to be a new setting and so the camera can film from either side of the action axis defined by the character's motion when it cuts to the room. The other dictates that consistent screen direction must be maintained from room to room.

2.5.3 Stationary Subjects

For stationary subjects, consistent screen direction means consistency of the relative positions of characters on the screen. For two or more characters the action axis is the vertical plane that joins the two characters that are closest to the screen and on opposite sides of the screen. Again, if the camera is positioned on one side of the action axis at the end of shot 1 it must remain within the 180º arc for the beginning of shot 2. This means

that characters will maintain their relative positions in-between shots. As before, the camera can move across the action axis during any shot.

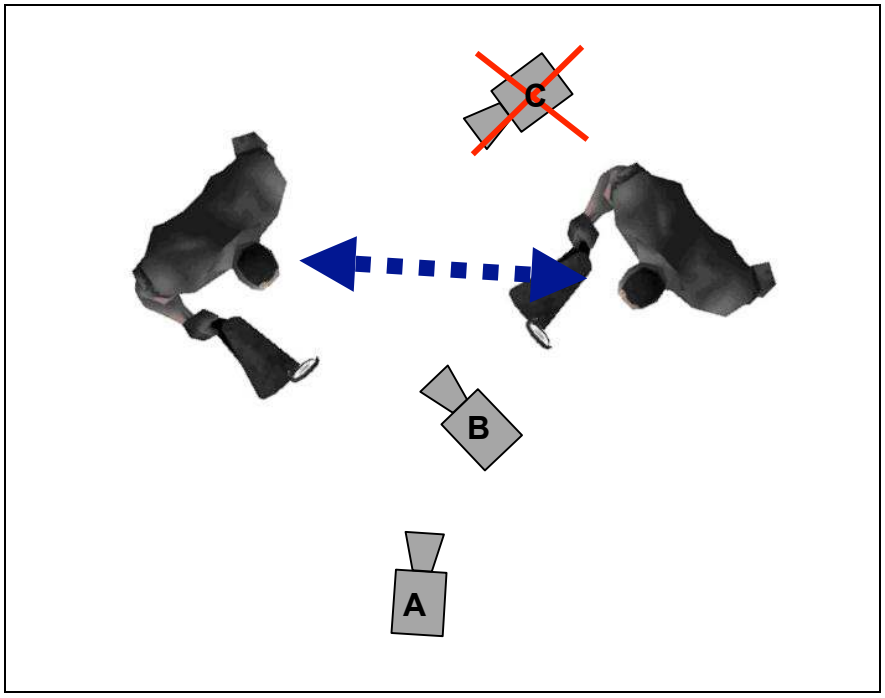

Figure 2.28: The action axis for stationary subjects. If the subjects are filmed with camera A at the end of shot 1, the camera must keep to this side of the action axis defined by the two subjects on opposite sides of the screen for the beginning of shot 2. Therefore placement B is valid while C is not

Continuity also applies to a character's look (Figure 2.29). A single character should look towards the same side of the camera from shot to shot for consistency. An action axis can be drawn in the direction he is facing and by adhering to this as before, the camera can shoot the subject from a variety of angles while his look will be consistent.

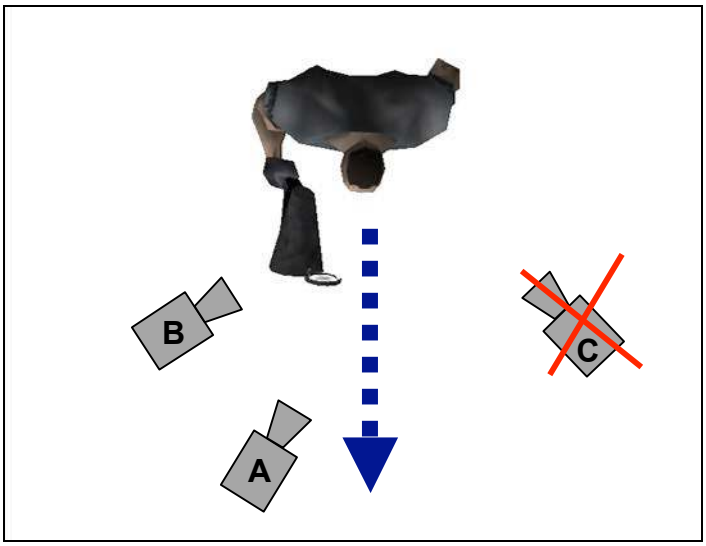

Figure 2.29: The action axis for a stationary subject. If the subject is filmed with camera A at the end of shot 1, the camera must keep to this side of the action axis defined by the subject's look for the beginning of shot 2. Therefore placement B is valid while C is not.

2.5.4 Reverse Shots

Reverse shots should also maintain continuity by adhering the action axis. A character might travel towards the camera and walk off the right edge of the frame before shot 1 cuts. Shot 2 can then depict the subject continuing on his journey by walking onto the left edge of the frame. This is achieved by rotating the camera after shot 1 to film the next part of the journey (Figure 2.30). If shot 2 is filmed from the other side of the action axis the subject appears to be moving in the opposite direction (Figure 2.31).

Figure 2.30: Reverse shot correctly filmed (the shot on the right is a reverse shot of the shot on the left): The camera has simply turned around for the second shot and so the subject appears to be continuing in the same direction

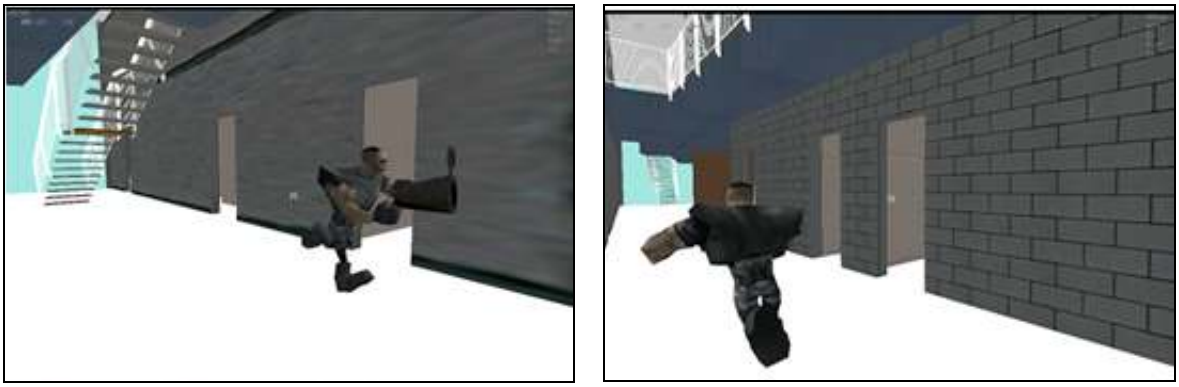

Figure 2.31: Reverse shot incorrectly filmed: The second shot was filmed from the other side of the action axis so the subject appears to be moving in the opposite direction even though he is not.

2.6 Cuts

As already mentioned, a cut is an instantaneous switch to a new camera placement or focal length. Cuts have numerous uses in cinematography including: to alternately depict the two participants in a conversation or to draw the viewer's attention to different parts of the scene. It is worth mentioning again that the notion of a cut is generally non existent thus far within the context of a computer game.

Mascelli (1965) suggests that a cut may only be carried out if the new shot represents a significant change in camera angle, camera placement, lens focal length or a combination of these. Failure to carry out a significant change results in an only slightly different shot called a jump cut which the viewer will find jarring and perceive as an editing mistake. A guideline to avoid jump cuts is the 20% rule (Figure 2.32 and Figure 2.33).

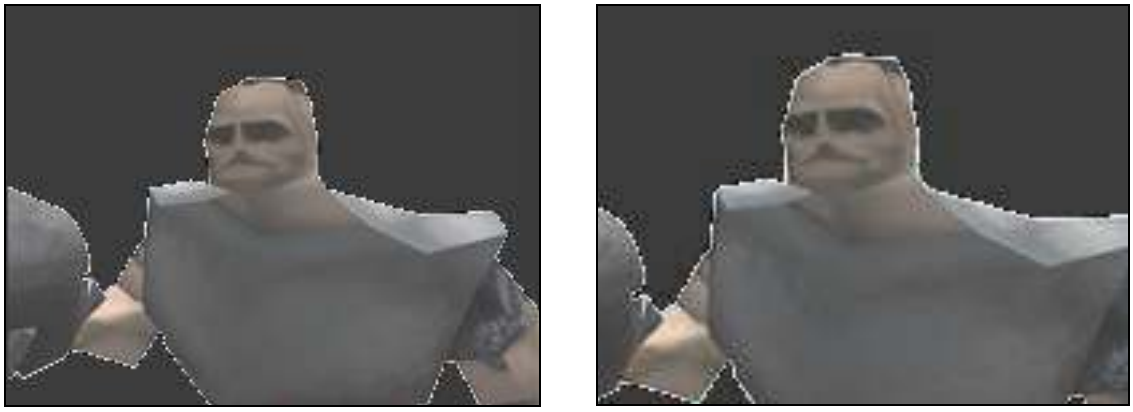

Figure 2.32: Jump cut: A cut from the shot on the left to that on the right does not conform to the 20% rule because the subject size has changed only slightly

Figure 2.33: Not a jump cut: The 20% rule is enforced and the change in subject size seems intentional

This means that if the distance to the subject or the focal length is to be changed for the next shot the change must represent at least 20% of its former magnitude. If it is the angle

between camera and subject that is to be changed, the new shot must place the camera at a rotation of at least 30° around the subject from its former orientation.

Specific types of cuts include *cross-cuts* and *cuts on action*. It is possible to film two events in an alternating fashion by cross-cutting. The two events will be occurring in either a different location or at a different time. Usually the first event will be filmed for a slightly longer duration of time before cutting to the second event and then each is shot in an alternating manner. Cutting on action applies to specific actions that occur in a setting. When filming action such as a person sitting down, it is common to cut as the action is occurring. In this case the character may begin to sit in a medium shot and then complete the movement in a close-up. Cutting on action is a useful tool for changing the subject size, camera angle and so on because the cut is smoother as it is disguised by the action. If the action is an important development in the story, however, it may be more suitable to allow it to occur in one shot.

Brown (2002) proposes some directives for cuts between moving and static shots, i.e. shots with a moving or static camera respectively. When filming a moving subject, these can generally be edited together. The subject could be filmed with a static camera initially, followed by a tracking shot and then another static shot. If two moving shots are to be edited together, however, the tempo must match. For a static subject, moving and static shots do not edit together well so moving shots should be eased into or out of the movement before and after static shots to make the transition more natural.

2.7 Scenes

There are a number of different ways to shoot scenes depending, not only on the style of the cinematographer but also on whether or not the action is *controllable*. Controllable action means that the director can arrange the scene and the placement of the actors and props within it for each shot and can film multiple takes of each shot. Uncontrollable action means there is no control to this degree as with a documentary of live events and, of course, computer games (this will be discussed in more detail in Chapter 4); the subjects cannot be given instructions and there is only one take.

2.7.1 Openings

A scene is commonly opened with an empty frame, i.e. a shot of an unoccupied setting before the characters enter, but it may also begin *media res*, i.e. in the middle of the action. Another common opening is to begin with a close-up of some object that is significant to the story and then slowly pull back to include the entire setting. This method, called *slow reveal*, can be used to create suspense where required.

2.7.2 Introductions

It is usually desirable to make some introductions to the viewer when beginning a scene. The first introduction to make is the location of the scene. If the action takes place inside a building, for example, an external shot of the building may be used. Generally the geography or layout of the scene will be introduced next with an establishing shot, e.g. a high angle long shot of the room containing the characters. Now that the viewer knows where the scene is set and has an idea of the layout of the scene, the principle characters are introduced perhaps with a full or medium shot. An exception to this approach is the slow reveal. In this case it is what the viewer does not know about the scene that is used as a narrative tool. Computer games have a specific type of introduction in the form of cut-scenes. These introduce the player to the setting before he gets to explore it with his avatar.

2.7.3 Master Scene Method

The master scene involves the filming of an entire scene with the *master shot* (a shot that includes the whole setting) along with *coverage*, i.e. full shots, medium shots and close-ups some of which may be filmed separately. This method therefore involves the use of multiple cameras. The footage from the master shot is edited with the coverage to produce a depiction with varying levels of subjectivity. *Plán Scene* is similar to the master scene method except that no coverage is filmed. The camera simply moves and pans around the setting to keep the action in view.

2.7.4 Triple take or overlapping method

The triple take method can be shot with a single camera and can only be used where the action is controllable. For this method the cinematographer always considers three shots: the current one, the previous one and the next one. The action ending in the previous shot is also shot at the beginning of the current shot and the action in the current shot has to overlap with that in the next shot. The reason for this is so that the editor has some duplicate footage from two angles to use when choosing a place for each cut.

2.7.5 Progressions and Regressions

Camera placement can be varied during a sequence so that the subject size becomes greater or smaller from shot to shot. Progressive or regressive shots like these may also be employed in which the area photographed increases or decreases from shot to shot; the image size must change by a minimal amount in such a sequence to make the change look intentional (see Section 2.6). Contrasting shots, i.e. pairs of shots with different image sizes, and repetitious shots, a series of shots of similar image size, are also common.

Progressive and regressive sequences are also possible in terms of the camera angle or viewpoint. A series of a progressively greater or smaller camera angles or camera heights may be used or the camera may move around from the front to a side or rear angle. Contrasting angles involve a shot of a high angle followed by a low angle or a front angle followed by a reverse angle. Overall, using the same sequences repetitively should be avoided; combinations should be filmed to maintain variety.

2.8 Conclusions

Of the various aspects of cinematography discussed above, we believe some are directly relevant to computer games and others not as much. The various types of shots, camera placements and lens types (Section 2.2) are readily applicable to games and even easier to carry out than on a film set for the aforementioned reason of a lack of physical constraints. The principles of composition (Section 2.3), on the other hand, are less suited to an interactive game since the action is uncontrollable and there is only a single take.

Those of framing might be more applicable. The many types of camera movements (Section 2.4) can certainly be carried out within a computer game and, as mentioned, with much ease. The movements and positions of characters in a game (a character's look less so) would definitely necessitate the use of continuity and conformance to the action axis (Section 2.5). The notion of a cut (Section 2.6) is a very significant departure from the traditional viewing system within games and breaks the mould of one level, one shot. We suggest that applying scenes (Section 2.7) to computer games would put each setting in context and give the player a proper introduction to it and its inhabitants.

Chapter 3

Related Work

In this chapter we move on to the use of virtual cameras, as opposed to real world ones. In order to produce a virtual cinematographic camera, it is necessary to automate a selection of operations derived from the principles and guidelines discussed in the previous chapter. This is a considerable task since usually the user controls the virtual camera in computer games and similar applications directly. A cinematographic camera control system would, therefore, have to remove this control to a greater extent. We examine a range of techniques that attempt to solve this and other problems relating to controlling these cameras in a number of different applications. We structure the discussion in terms of four categories of virtual cameras in increasing levels of sophistication – static camera, moving camera without subject, moving camera with subject, and cinematographic camera – and highlight works that have influenced our own. The typical tasks involved in implementing these configurations include positioning and orientating the camera, preventing occlusion of the required view, moving the camera smoothly and, of course, storing and applying guidelines from cinematography.

3.1 Static Camera

The most basic camera configuration we discuss is the static camera. This is one that takes up a position and orientation and does not move for the duration of its filming. Many of the shots discussed in Chapter 2 such as the long shot, medium shot and close-up are often filmed as static ones and so require this kind of camera. There are two methods for positioning and orienting a camera within a virtual environment. For the first it is necessary to define the three components of its position in three-dimensional space,

the three components of its orientation and its field-of-view. For the second method the camera's placement is defined with the `lookFrom`, `lookAt` and `up` vectors (Blinn 1988). The `lookFrom` vector defines the camera's position; the `lookAt` vector the location towards which it is to be oriented; and the `up` vector its positive vertical axis. The methodology for both of these approaches is covered in Foley et al. (1990). Having chosen one, the task is now to instruct the camera system to include specific scene elements (characters and props) in the view and to use particular types of angles and/or subject sizes as required. Given this information it is up to the camera system to ensure that a clear view is presented.

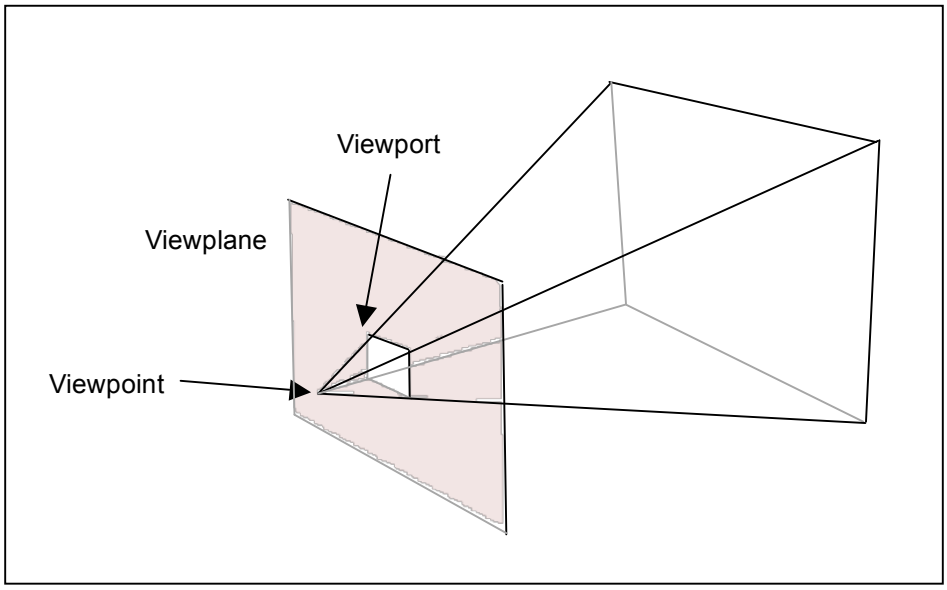

Figure 3.1: The viewpoint, viewplane and viewport

Halper & Olivier (2000) use a genetic algorithm (Luger & Stubblefield 1998) to automatically generate the optimal camera placement within a virtual environment given some *shot properties* as defined by the user. These include the required position, size, orientation and visibility of scene elements (characters and props) relative to the *viewplane* (the plane that is perpendicular to the viewing direction), *viewport* (the section of the viewplane that is viewed) and *viewpoint* (the location of the camera). See Figure 3.1. It might be required, for example, that a scene element is located between two points specified in the viewport, that it is oriented at a certain angle to the viewplane or that a certain proportion of it is visible in the viewport.

The required properties of a given scene element can also be specified relative those of other scene elements as they appear in the view. The user may dictate that one scene element appears twice as large as another, to the left of another or is only partially occluded (to a specified percentage of its entire visible area) by another. In defining shot properties, the user can specify tolerance settings so that the system can give preference where a conflict between requirements arises.

With the shot properties defined, the optimal shot is found by a genetic algorithm. Genetic algorithms model the solving of a given problem as a "competition among a population of evolving candidate problem solutions. A 'fitness' function evaluates each solution to decide whether it will contribute to the next generation of solutions. Then, through operations analogous to gene transfer in sexual reproduction, the algorithm creates a new population of candidate solutions" (Luger & Stubblefield 1998). Halper et al. encode the seven elements of the camera state vector, that is, those of its position vector, those of its orientation vector, and its field-of-view, in the chromosomes of each gene and randomly distribute a population of these in a bounded search space. The overall fitness value is derived from a combination of the fitness values of all of the genes based on their adherence to the shot properties defined by the user. In determining these fitness values the system evaluates the following image properties:

- The location of the centre of a scene element.
- The projected extents of scene elements.
- The visibility of scene elements.

To maximise the accuracy of performing evaluations while minimising the cost, an evaluation window is used. The evaluation window encloses only the pixels of the view containing the image properties to be evaluated. The resolution of the evaluation window, and therefore the cost of evaluations, is determined by the dimensions of the viewing window and the tolerance settings defined by the user. By keeping it as low as possible within the user's tolerance settings the cost of calculations can be kept to a minimum.

After evaluation the fittest 90% of the population of genes survive into the next generation. The other 10% are regenerated by random crossover and/or mutation of the genes. Generations continue to be produced until the user aborts or an optimal combination of chromosomes is found for the camera state, i.e. the evaluation of the camera state has determined it to best conform to the user's requirements. Although this is a sophisticated method for producing an appropriate shot given some visual requirements, it is not designed to operate in a real-time application and so is not suited to our requirements.

3.2 Moving Camera without Subject

The second type of camera configuration we discuss is a moving camera without a subject. This is a camera that moves through a setting with no subjects to film and it might have one of two implementations:

- One is the first-person perspective used in FPS games where the view presented is that of the avatar. In this case the camera is directly controlled by the player in that its movements are matched to those of the player's avatar. As discussed at the beginning of this chapter, one of the main problems to solve in implementing a cinematographic virtual camera is to remove this control.
- The other is a moving view that is not necessarily that of any character in particular and is used for architectural walkthroughs.

The problems to solve in order to successfully implement such a camera configuration include placing the camera at valid positions in the scene and ensuring the view is clear as it moves. The design of a first-person camera is relatively simple task in that it is simply a case of granting control to the user. For the purposes of our work, it is not necessary to carry out this design because the camera is readily available in all FPS game engines. The walkthrough implementation is not suited to the main task of our work but it is important to consider as a progression from a static camera towards a cinematographic camera and might have applications secondary to our work so we consider an approach to its implementation here. Nieuwenhuisen & Overmars (2003) implement such a camera with the probabilistic roadmap method (Švestka 1997). Given the camera's current

position and a goal position entered by the user in real-time, the system generates a path between the two for it to follow.

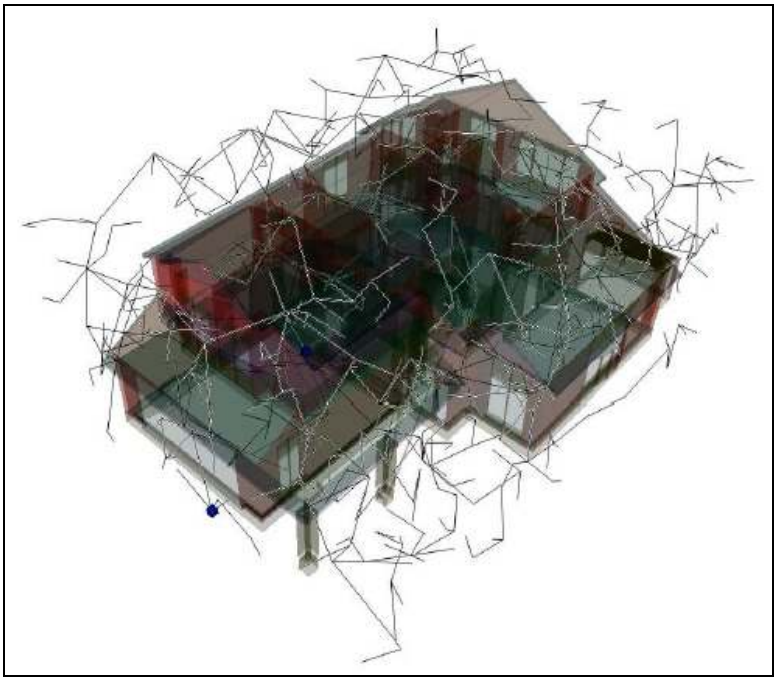

Figure 3.2: An example of a roadmap for a building

In order that the system can find a suitable path in a reasonable amount of time a probabilistic roadmap is created in a pre-processing stage. A similar process was considered for our own work for the detection of rooms in a computer game (see Section 4.4) but was later discarded. Nieuwenhuisen & Overmars' roadmap represents a number of valid camera placements in the virtual world and clear paths between valid placements (Figure 3.2). Each node in the roadmap represents a valid placement and edges between nodes represent valid paths and so providing a moving view through the environment is a case of finding a route through the roadmap. When the camera travels from one location to another there must be sufficient clearance from objects in the scene so that there are no collisions or near collisions. It is therefore represented as a sphere whose radius determines this clearance. The roadmap is created by positioning a sphere of the same radius at randomly selected locations in the scene. If the sphere does not intersect with any objects in the scene it is added to the roadmap as a new node. Paths between nodes

are generated by placing a cylinder of the same radius between pairs of them. Again, if the cylinder does not intersect with scene geometry it is added to the roadmap.

To generate a path from a starting point to a goal position in the scene, two nodes are created for both locations and connected to nodes in the roadmap. If the starting node and goal node exist in the same connected component of the roadmap a path exists between them. However, if either node cannot be connected directly to the roadmap, more nodes can be added to the roadmap in the vicinity of the unattached node. If both nodes are connected to the roadmap but are not in the same connected component the roadmap may be expanded by increasing the node density. The shortest path between the start and goal nodes is found using Dijkstra's shortest path algorithm (Dijkstra 1959) in conjunction with a penalty function for paths that are too long or have sharp corners. The path is made smooth by applying circular blends to corners with an arc of the largest possible radius but which still ensures clearance of objects in the scene. When the camera travels along the path it slows down at bends in the path and speeds up for straight sections. This is facilitated by a speed function that analyses the path and returns a speed diagram for the optimal speed for each point on the path. The orientation of the camera is set such that it always looks at its location a unit of time in the future (usually one second) rather than in its immediate direction of travel. This has the effect that when it moves into a bend it will look further along the bend giving the user better orientation.

The first-person perspective camera is part of a set of many other types of shots in cinematography as discussed in Section 2.2 and so should be considered for use in a cinematographic virtual camera. As stated previously, a walkthrough camera is not directly relevant to our work, i.e. filming the main action of a shooter game, but it does have an application in other computer games, e.g. Prince of Persia: The Sands of Time (Ubisoft 2003), to automatically generate a walkthrough of a game level in order to serve as an introductory device.

3.3 Moving Camera with Subject

The third type of camera configuration we examine is the moving camera that includes a subject in its frame. Such a camera can be considered to provide a third-person

perspective in that it doesn't provide any character's view in particular. This configuration is used in a large number of computer games with the exception of first-person shooters, the genre of game that we have chosen to modify. The type of camera angle (see Section 2.2) used in conjunction with the third-person perspective varies from game to game but a popular type is a rear shot of the avatar. Here the camera moves closely with the avatar and is usually positioned above and behind the avatar's head such that the avatar is positioned at the bottom of the screen (e.g. Tomb Raider III (Eidos Interactive 1998) and Hitman 2 (Eidos Interactive 2002)) – see Figure 3.3.

Figure 3.3: Third-person camera in Tomb Raider III

The approach we discuss here to the implementation of such a camera is that of Halper et al. (2001). The authors present *A Camera Engine for Computer Games* in which *geometric constraints* on the camera are the basis for control. Specifically, constraints are defined on the position and orientation of the camera relative to the subject or the scene and are updated as the action ensues. The principal problem to solve here is to automatically reposition the camera continually so that it follows its subject in the fashion required by the application. Firstly constraints are specified on the degrees of freedom of the camera. These might be the three components of its position, the three components of its orientation and its field-of-view, or the `lookAt`, `lookFrom` and `up` vectors depending on the placement method chosen (see Section 3.1) for a given approach. Then the camera control system applies these constraints to the camera on a continual basis to produce suitable views within this framework.

One of the main issues the authors address is frame coherence, that is, ensuring that the transition from one camera configuration to the next is smooth and coherent. The alternative is a best-fit solution by which the camera is instantly repositioned at the new ideal position every time the scene content changes thereby producing erratic movements. Halper et al. achieve smoother movements and transitions at the expense of less-than-perfect adherence to geometric constraints. They also suggest that a camera engine should be part of the computer game pipeline following the calculations of game logic, AI and user interaction, and preceding those of lighting and rendering.

The camera system has two main elements: The *director* and the *predictive camera planner*. The director selects appropriate *shot templates* (descriptions of the set of constraints appropriate to each type of camera configuration) from the *shot library* based on the list of actions in the previous cycle of the game. *Emotion templates* tweak these constraints to create different moods. The director uses transition rules to ensure that one configuration transitions correctly to the next. The predictive camera planner uses the present and past locations of scene elements to predict the future state of the scene in order to plan camera placements.

An element of the predictive camera planner called the *constraint solver pipeline* applies a number of different constraints to the camera, one after the other, for each configuration of the camera. Each constraint is given a weighted value, e.g. size 30%, so that if a conflict between constraints occurs, those with the weakest weights can be relaxed. Frame coherence is implemented by:

- Applying constraints according to the previous camera configuration
- Allowing constraints to be relaxed where appropriate to facilitate frame coherence
- Using *lookahead* algorithms to predict the future state of the scene

The pipeline solves the constraints in an order such that each has a minimal effect on the solution from the previous one. There are a number of constraints including that on the height of the camera relative to the subject, the side of the subject that is to be filmed and the size of the subject in the view.

A camera that follows a moving subject is an essential tool for any cinematographer (see Section 2.4) and directly applicable to the filming of shooter games for parts of the game where the player is navigating as opposed to fighting.

3.4 Cinematographic Camera

Our discussion now comes to the type of virtual camera that our own work addresses, the cinematographic camera. The term cinematographic camera with respect to our discussion of related work refers to one that employs varying levels of subjectivity (by the use of a variety of camera angles); presents a number of different types of shots (and therefore incorporates the notion of a cut) of a given scene; takes continuity into account; and follows some guidelines for the shooting scenes appropriate to the action as discussed in Chapter 2. To implement a suitable camera control system two main issues must be resolved.

1. The system must have the ability to acquire sufficient information about the virtual scene to film the required shots

2. The system must store the cinematic guidelines: camera configurations for different shots, the timing of cuts and so on

We now discuss works that address these issues.

3.4.1 Film Idiom-Based approaches

A film idiom is a standard method of applying a number of shots to a given scene (Christianson et al. 1996). For example, given a scene of two characters facing each other in a conversation, a standard approach is to begin with a two-shot, then over-the-shoulder shots of each character and finally close-ups of each character. The view often moves back out to the two-shot and then inwards again to the close-ups as the dialog ensues. Sometimes the speaker is filmed and sometimes the other character's reaction is filmed. The difference between using film idioms and using general approaches to shooting scenes like those discussed in Section 2.7 is that film idioms are catered for scenes containing a specific set of actions. Obviously a drawback of this is that the course of

action must be known in advance, which rules out the general application of film idioms to virtual cameras in computer games. Nevertheless it is important to consider this approach as it is used in many similar applications.

Christianson et al. (1996) use film idioms for an off-line approach to depict 3D animations. Their *Camera Planning System* (CPS) uses an animation trace of action that has occurred previously to generate the cinematic footage. In order to do this it separates the trace into scenes and applies a suitable idiom to each one. Idioms are defined with their *Declarative Camera Control Language* (DCCL). An idiom definition specifies the shots to be used for a given type of scene and Christianson et al. found it necessary to further decompose shots into *fragments* where a fragment represents a simple camera movement or a static camera. There are five types of fragment in DCCL including static, panning and tracking camera fragments.

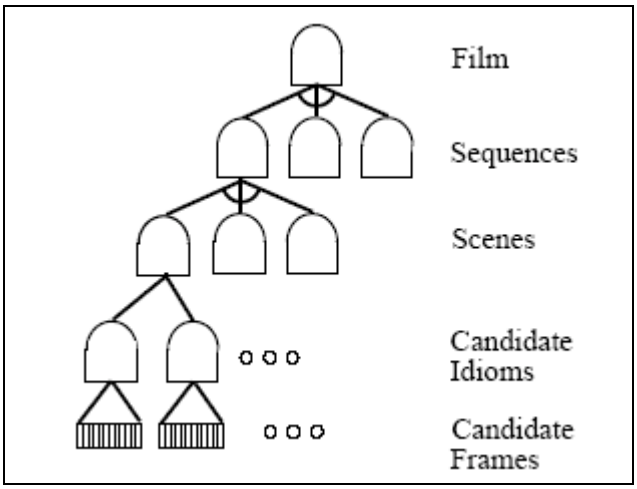

Figure 3.4: The film tree

The CPS consists of three modules: the sequence planner, the DCCL compiler and the heuristic evaluator. Each module adds information to a hierarchical structure called a film tree (Figure 3.4) starting with the sequence planner, the only domain specific module in the CPS. It defines the film tree down to the scene level and in order to do so must analyse the animation trace. The animation trace is defined as one or a number of parallel film sequences of one or more settings. These sequences define the actors that are to be filmed over a particular time period, and the actor that is to be the protagonist. The

sequence planner firstly partitions each sequence into scenes according to the activities being performed, and then applies suitable idioms to each one. If a number of sequences cover the same time frame, multiple candidate idioms will be produced.

Next the domain-independent DCCL compiler expands the candidate idioms into specific camera configurations for each frame thereby defining the film tree down to the level of candidate frames. These are analysed by the heuristic evaluator which is also domain-independent in that it ranks the quality of candidate idioms on the general standard of camera work. It uses a scoring mechanism to decide which idioms to keep and which to remove from the scene tree. Successful idioms must maintain smooth camera movements and not cross the action axis. Fragments that are too short or in which the camera pans backwards will be discarded. In this way the heuristic evaluator chooses the best idiom for each scene. It then concatenates the frame arrays for all idioms remaining in the scene tree and outputs the frames for playing. Our own implementation differs from Christianson's in that we do not use film idioms and ours is in real-time but the fundamental application of cinematic shots is the same.

He et al. (1996) apply cinematic idioms to a real-time scenario, namely a real-time virtual party setting. Users take part in the interactive virtual party over a computer network by typing commands to instruct their protagonist to take part in a conversation, walk to a location in the room or order a drink at the bar. Each user has his own *virtual cinematographer* (VC) which portrays the scene relative to his protagonist. The three main components of the system are the real-time application, the VC and the renderer. At each time tick the real-time application sends data to the VC describing events that occur in the virtual party relative to the user's protagonist. The VC then produces a shot specification based on these events and the existing state of the animation. It may query the real-time application for additional information such as the location of actors in the virtual party and for some shots may reposition actors subtly for optimal placement on the screen. It sends this information to the renderer as *acting hints* along with the shot specification. The real-time application sends details regarding the state of the environment and animation parameters to the renderer. The renderer then renders the scene.

The VC architecture consists of two main components: *idioms* and *camera modules*. Idioms contain the directives for capturing different types of scenes; camera modules describe specific shots and are combined to produce idioms. There are sixteen different camera modules in the VC architecture, mostly for filming characters in conversation, including a shot of two actors on opposite sides of the screen, over-the-shoulder shots and tracking shots.

The camera modules enforce adherence to the action axis and detect occlusion. At each time tick if there is any occlusion a counter is increased. The counter is reset to zero if the actors that were occluded are no longer and the idioms can change to a different shot depending on the value of the counter.

Idioms are arranged in a hierarchy with a master idiom at the top and more specific idioms towards the bottom. This means that the VC can return to a more general idiom if events unfold that the current one cannot account for. Each idiom is constructed as a finite state machine (FSM) in which each state represents a different camera module. If certain conditions are fulfilled the current state is exited along an arc to another state. In this way the idiom encodes the methods for applying various shots to different scenes. Again, our cinematography system is distinct from that of He et al. since we do not employ idioms and ours is catered for shooter games and so does not include shots of conversations, however we do use an FSM to control editing decisions and model the game world activities.

Amerson and Kime (2001) present an approach to controlling virtual cameras through cinematography as an attempt to harness the story-telling capabilities of cinematography for real-time interactive narratives in virtual worlds. They store film idioms in a *scene tree*, a hierarchical structure similar to that of He et al. At the root of the tree is a generic type of idiom; at the next level, nodes are differentiated by shot type; the next three levels are differentiated by the number of participants in the shot, its desired emotional effect and a list of keywords for fine tuning idiom selection. The idioms are encoded using Amerson and Kime's FILM (Film Idiom Language and Model) system. Within this system, each idiom contains a number of shots and each shot contains a number of parameters such as the event that triggers a cut to the next shot, the list of constraints to

control the camera's position relative to the subject for the shot and so on. Each constraint has a weight associated with it to define its importance to the shot.

E.g. lensType(NORMAL_LENS) WEIGHT=.9

This defines the constraint on the type of lens to have a weight of 90%. The components of the interactive narrative system are the game engine (the Unreal Tournament game engine) and the AI server. Within the AI server is the *narrative planner.* This generates the narrative and sends it to the game engine. The games engine then communicates the user's actions in the world back to the AI server. If the user's actions interfere with the narrative planner's plan, steps are taken to either alter the action or the plan itself.

The camera control architecture consists of the narrative planner, the *Director* and the *Cinematographer*. For each scene, the narrative planner sends its plan to the Director. The Director uses these scene requirements to select a scene from the scene tree by performing a depth-first search with no backtracking. The deeper the Director searches the more refined the idiom is but it will not search any deeper once the requirements are fulfilled since this will result in communicating superfluous information to the user. After binding variables in the idiom to objects in the virtual environment, such as the actor to film, the Director passes the scene specification to the Cinematographer.

The Cinematographer chooses the optimal camera placement based on the constraints defined in the idiom. If a suitable placement cannot be found due to occlusion of the view or a conflict between constraints, the Cinematographer can relax one or a number of constraints, usually those with the weakest weights. An ad hoc approach to relaxing constraints is used in which predefined procedures adjust camera parameters. When this is complete the Cinematographer sends the camera specification to the graphics engine for rendering. Again, our approach is similar to this one in some ways but not in others: We also use a Cinematographer module but not a Director, and, again, we use a non-idiom approach.

Ting-Chieh et al. (2004) seek to enhance the atmosphere of computer games and enrich the gaming experience by applying techniques from cinematography to the camera. They

formalise various concepts such as the action axis and establishing shots, and encode them in the *cinematic camera control system*. The system architecture consists of a real-time application, a cinematic camera control system and a renderer. At each time tick the real-time application sends parameters to the cinematic camera control system such as actors' positions, orientations, equipment, activities and user input. The cinematic camera control system generates a camera specification based on these parameters and the current state of the camera and outputs this to the renderer. The renderer uses the camera specification and a description of the environment sent by the real-time application to render the scene.

The cinematic camera control system consists of a number of *camera modules* and *descriptions of shots*. Descriptions of shots, which are equivalent to idioms as used by Christianson et al. and Amerson and Kime, are arranged in a hierarchy with more general descriptions closer to the root. For example, at the first level of the hierarchy there is a shot description for a conversation. Its children nodes are *two talk* and *three talk* for two-person and three-person conversations respectively. Camera modules are lower level descriptions for filming individual shots and can adjust the camera state and constraint lists to alter the shot produced where needed. The available constraints include the height of the camera relative to the subject, the camera's orientation relative to the line-of-action (action axis) and the camera's orientation relative to the direction a character is facing.

Camera modules ensure that frame coherence occurs, convert any jump cuts (see Section 2.6) to a gradual change in the camera specification, such as a smooth movement to the new location, and ensure that the camera does not cross the action axis. Occlusions are prevented by testing if a cylinder drawn between the camera and subject intersects with scene geometry. If intersection occurs, the camera may move back along the direction it came to provide a clear view.

In the game system itself the user may partake in a number of activities such as holding a conversation, exploring the environment, taking part in a gun fight or a close fight. Each of these activities corresponds to a shot description which is itself a finite state machine (FSM). Each state in the FSM represents a camera module. For example, the states in the

exploration shot description are point of view, fixed, track and pan and so these are the four types of shots used when the user is exploring. Specific events will trigger a transition between states and therefore a cut to a new shot. This implementation and specifically the area of application are quite similar to our own in that there is an emphasis on camera modules for fighting actions. Again, we did not develop for conversation shots as these do not occur in Quake II.

Charles et al. (2002) use an idiom-based approach in the presentation of interactive narratives. Events in a given narrative are represented as syntactic triples, i.e. subject, verb and object, where the subject is an active character and the object is another character, an object in the scene or null. Examples of verbs are *move*, *pickup*, *talk* and *idle*. Each triple has a *story weight*, i.e. its importance to the narrative. The system uses these weights to create a heuristic classification of the active events and then to choose the most significant one. A suitable idiom is selected for this event and applied to the virtual world. Idioms are, again, modelled as finite state machines where each state represents a different shot. For example, the idiom for three characters talking has the states:

1. Establishing shot of all three characters
2. Shot of characters A and B talking (this is based on the idiom for two characters talking)
3. Shot of character C talking
4. Shot of character C reacting

The camera control system continually evaluates the view to account for occlusion and will modify the camera configuration if necessary.

3.4.2 Non Idiom-Based approaches

Funge (1999) uses a *cognitive modelling language* (CML) to create artificially intelligent characters for computer games. CML defines a character's knowledge about the world they inhabit, how they acquired that knowledge and how they can use that knowledge to

plan actions. The authors use CML for, among other things, a cinematic camera controller for computer games.

Cognitive modelling is broken down into two subsets: *domain knowledge specification* and *control specification*. Through domain knowledge specification the character is granted knowledge about the world and how it can change. The *situation calculus* (Levesque et al. 1998) is used to describe the changing world as a sequence of situations, i.e. snapshots of its state, and any property of the world that may change over time is known as a *fluent*. An example in the camera controller implementation is the fluent `Talking(A,B)` which is true whenever characters *A* and *B* are having a conversation. Fluents change due to characters performing actions and these may only be performed within restrictions imposed by the user. In the case of the fluent `Talking(A,B)` the camera controller can only perform the action of filming an external shot (over-the-shoulder shot) of character *A* if already filming *A* and it has not becoming boring yet or if not filming *A*, *A* is talking and the current shot has lasted for long enough.

The user also specifies the effect that an action has, such as character (in this case the camera) adopting a new position in the world. Characters are to assume that these user-specified effects are the only ones that occur in the world, i.e. the world stays the same unless otherwise indicated. The action `external(A,B)`, for example, results in the camera being directed at character *A* and being located above the shoulder of character *B*.

The second aspect of CML, control specification, is needed to direct a character to behave in a certain manner because the domain knowledge specification alone is unlikely to be enough for the character to generate a plan of action unassisted. To narrow the search space such that a more specific plan will be generated the user can write complex actions in CML. A complex action is an action constructed of other actions by using regular programming constructs, such as loops. The following statements instruct the camera controller to shoot a conversation appropriately:

```
setCount;

while(0<silenceCount){
```

```
        pick(A,B)external(A,B);

        tick;

    }
```

In particular these statements direct the camera controller to pick a new external shot (over-the-shoulder shot) of either character A or B after a certain amount of time has elapsed. Detailed instructions can be omitted from the control specification because *reasoning engine* can fill them in at run-time. An advantage to this overall method is that it is possible to create a loosely defined working prototype initially and add to the control specification later in order to narrow the focus of the reasoning engine and make the character act more appropriately. We decided that the state of the Quake II game world did not justify the use of CML and to use finite state machines instead.

Tomlinson et al. (2000) propose an automatic cinematography system for interactive virtual environments that "makes it easier for participants to interact with [the virtual] world and displays the emotional content of the digital scene". The camera control mechanism in this case is their *CameraCreature*, one of a number of artificially intelligent (AI) entities that populate the system. The CameraCreature controls the camera and positions lights in the scene. Through its *sensors* it assesses the *emotions*, *motivations* and *actions* of the other characters and combines this information with its own internal state to determine what actions to take. Its emotional state is determined by that of each character in the scene and this, in turn, determines its shooting style. A happy camera may cut more frequently and use oscillating movements whereas a sad camera may have long swooping movements. The amount of influence a particular character has determined by the amount of screen time they have had recently, how important they are and the intensity of the emotion they are feeling. Motivations affect filming in a more localised way. For example the default motivational state of the CameraCreature is to try to frame two characters having a conversation.

At each clock tick the CameraCreature must decide what actor to film (it usually chooses the actor controlled by the user), where to place the camera and what lighting to use. The

placement of the camera with respect to the actor can portray a wide establishing angle, a navigational angle (so the user can see where the character is going) or perhaps a close-up (to express more emotion). The CameraCreature uses different transitions to get from one shot to the next such as cuts and whip-pans (these cause the camera to swoop through space to the new position). To detect occlusion the system casts a ray from the camera to the subject. In one implementation the camera accounts for occlusion by moving up until the line-of-sight is clear.

The communication and evoking of emotions are some of the ultimate goals of cinematography. However unlike Tomlinson's work, they are not modelled in shooter games and the emotion that should be experienced by a specific character at a given instant might only be inferred by the actions occurring in the scene.

Friedman and Feldman (2002) see idioms as being too coarse as a means for formalising cinematic principles and instead choose fine-grained rules for this purpose. Like Christianson et al. (1996) their work is applied to offline animations. In particular, their system is supplied with an annotated screenplay and a raw animation and to this it applies rules from a knowledge base to produce editing decisions. If it is supplied with 3D meshes and animations it also generates 3D footage. The screenplay is written in a formal language as in the example below:

```
location: living-room
init: Mario sit sofa1
Mother enter room
Mother speak "Mario, have you
eaten the sandwich I made
you?"
Mother sit-on sofa2
Mario speak "I wasn't hungry."
```

Each screenplay is converted into an abstract representation of action frames and spatial annotations which are placed on a time line. Then scenes in the screenplay are divided into *sentences*, visual units smaller than scenes but greater than shots: A new sentence occurs due to a change in some specific action the definition of which depends on the

cinematic genre. Now rules from the knowledge base are converted into constraints and matched up with the corresponding frames. For example, one rule dictates that the first or second shot of each sentence must be an establishing shot and an establishing shot must frame the scene in a long shot. Another example is 180-degrees-line rule (the conformance to the action axis):

```
(for-all I (intervals this-sentence))
➔
(= (side (shot I) 180-line)
(side (shot (next I)) 180-line))
```

This means that the camera has to be on the same side of the action axis before and after a cut. As constraints are applied, it is possible to back track so that earlier constraints facilitate those that come after. In the case of the 180-degrees-line rule, therefore, it might be the camera placement before rather than that after the cut that is changed to conform to the rule. When the footage is produced, the user can over-ride any constraint in particular by applying her own. This change is propagated throughout the set of constraints applied and a new solution presented.

3.5 Conclusions

We now conclude on the applicability of the different types of camera configurations discussed above to our work specifically. As discussed earlier, many cinematic shots are carried out as static ones (Section 3.1) so the static camera configuration is essential for creating a virtual cinematographic camera. However, the use of a genetic algorithm by Halper et al. is unnecessary for our purposes and we instead use simpler algorithms such as iterations of increasing subject angles, subject distances and so on to find a clear shot rather than a random distribution of varying camera configurations. The moving first-person camera configuration (Section 3.2) is one of many used in cinematography and is still relevant to a virtual cinematography system for shooter games even though an overhaul of the camera system is to be performed. The walkthrough camera might have an application as an introduction to a level and is proposed as future work.

The idiom approach (Christianson et al. 1996; He et al. 1996; Amerson and Kime 2001; Ting-Chieh et al. 2004; Charles et al. 2002) as discussed in Section 3.4.1 is not generally suitable for shooter games since the course of action must be known in advance when using idioms, and the works above have used them to a greater extent to film conversations which generally do not occur in shooter games. The use of finer grained rules as proposed by Feldman and Friedman (2002) is more applicable to shooter games. We see the use of a cognitive modelling language (Funge 1999) as unnecessary for our purposes and favour instead finite state machines as used by Ting-Chieh et al. (2004). Although one of the principal aims of cinematography is to portray emotions, those of a given character are not readily obtainable from a shooter game in the same way as they are in the work of Tomlinson et al. and must be inferred from the action if required. It is worth noting that the more sophisticated camera control methods are carried out off-line and so are not applicable to our real-time implementation.

If one is to create a cinematographic camera for computer games, it is necessary to consider any commercial work that has already been carried out in this respect. Of the camera configurations discussed in this chapter, the first-person moving camera and the third-person moving camera positioned at the rear of the avatar are already used in a number of games and so one would not need to design these when adding cinematographic capabilities to a given game's camera system. In addition to these, some games insert cut-scenes between sections of normal game play. One game that we would like to draw particular attention to in its use not only of cut-scenes but also of varying camera configurations and the removal of user control of the view is Prince of Persia: The Sands of Time (Ubisoft 2003). Here there is the use of varying subject distances according to the task at hand and also the insertion of cut-scenes between levels.

Chapter 4

Game Cinematography System

In this chapter we present the components of our virtual cinematography system and outline their operation. We also discuss our use of the principles and guidelines of cinematography discussed in Chapter 2 and our approaches to general camera control problems discussed in Chapter 3.

4.1 Overview

In Chapter 2 we introduced the roles of the director, cinematographer and editor on a real-world film set. In addition to these there are camera operators who carry out the physical tasks of moving cameras around, adjusting the focus and so on. Our system is modelled on all of these roles although we decided to roll the director, cinematographer and editor into one software representation, the `Cinematographer`. This means that our `Cinematographer` takes on the action-selection role of the director and the cutting role of the editor in addition to coordinating the camera operators. This virtual `Cinematographer` and our virtual camera operators carry out their work in the Quake II game world. This game provides a first-person view which is directly controlled by the player, so issues of good visibility of the scene and so on do not factor in because it is up to the user to control the camera. Our work moves away from this restricted view by providing a number of different types of shots and selecting the appropriate one to use at a given instant based on (a) the action occurring in the game and (b) cinematography principles. The player is granted more general views of the game world when their avatar enters a new room for example and special shots for combat situations. These shots are provided with cinematic devices including freeze frame and slow motion. We discuss the types of shots, cinematic devices and editing decisions provided by our system later in this chapter.

In Chapter 3 we highlighted the two fundamental problems that must be solved in order to implement virtual cinematography:

1. The system must have the ability to acquire sufficient information about the virtual scene to film the required shots

2. The system must store the cinematic guidelines: camera configurations for shots, the timing of cuts and so on

Our system solves the first problem by communicating with the game via a well-defined interface that we have developed ourselves (this is discussed in the following chapter), and representing the action in the scene with a finite state machine (similarly to the work of Ting-Chieh et al. (2004)). We did not see the necessity to store film idioms in a hierarchical structure like Christianson et al. (1996) and He et al. (1996) in order to solve the second problem, but instead, like Friedman & Feldman (2002), chose to store finer-grained rules within the structure of an FSM used to represent the action in the game. The FSM will be discussed in Section 4.3.

Cinematography tells us that when filming any subject matter the nature of the content must be considered as discussed in Section 2.7. If the action is controllable as is the case with motion pictures, multiple takes may be filmed and only one camera is needed. With FPS computer games on the other hand the action is not controllable. It is played out by a number of computer-controlled characters and the player (our system is for single player games only, for reasons that we outline in Section 4.3) in real time. There can only be one take of a given shot so cinematography dictates that multiple cameras must be used to provide the maximum choice of footage for editing decisions. In accordance with this principle we created a number of different types of camera operators suited to different tasks with each one carrying out a different type of shot (Section 2.2) to maximise the editing potential.

Our first three aims, as presented in Chapter 1 were to develop a virtual cinematography system capable of:

- Providing more relevant views of the game content that are not limited to the player's point of view
- Varying the level of subjectivity throughout the game
- Adding dramatic emphasis where necessary

In order to do this it was necessary to expand upon the single type of shot granted by FPS games, i.e. the first-person perspective. The first-person perspective was still considered to be an important shot, but one of a set provided by our system. In order to decide what types of shots to add to the game we considered the guidelines for shooting scenes. The principle of introductions (again, see Section 2.7) offered a universal way of coordinating multiple cameras such that the viewer is granted varying levels of subjectivity, dramatic emphasis where necessary (see above), and a better chance of finding her bearings within a game setting.

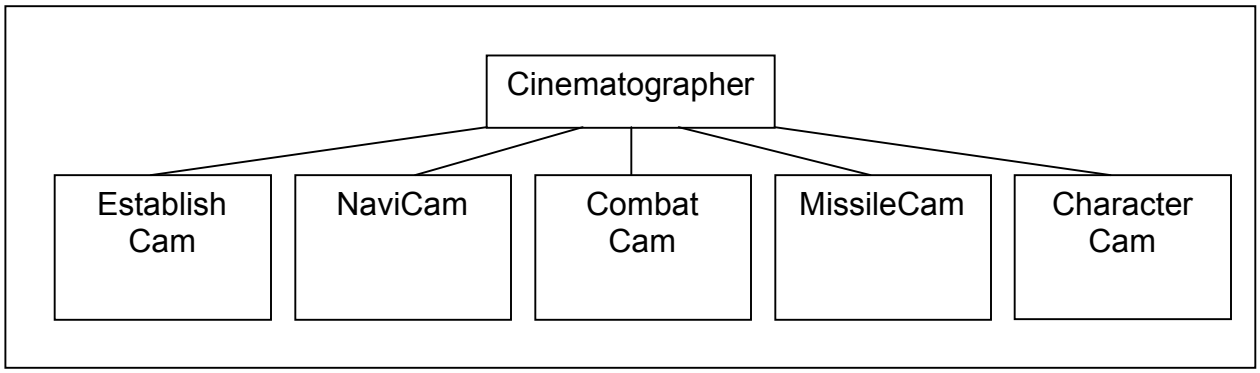

Figure 4.1: Virtual cinematography system hierarchy for shooter games

Our choice of shot types and therefore camera operator types takes into account both this principle and the tasks the player has to carry out in a FPS. The actions of the camera operators must be coordinated in such a way that these considerations are accounted for and we do this with our cinematographer. The cinematographer also takes on part of a film editor's role in deciding which camera is to provide the player's view at each instant. The editing decisions are also based on the principle of introductions, the tasks the player carries out within the context of the game and, of course, editing principles. Figure 4.1

illustrates the relationship between the cinematographer and the camera operators. These are discussed in the next section.

4.2 CameraBots

Our camera operators are implemented as invisible *bots*. Bots (Fairclough et al. 2001; Laird 2000; Laird & Duchi 2000; Reynolds 1999) are autonomous entities that can freely move about a game world. It is possible to give bots specific instructions regarding their position and their movement in such a way that they mimic the behaviour of camera operators on a film set. The game engine can then present the action from the viewpoint of any bot in particular. Specifically, bots are capable of taking up a position and looking in a certain direction; they can find a path through a virtual environment that avoids obstacles (Graham et al. 2003; Graham et al. 2004); they can crouch, turn, follow a subject, approach a subject and retract from a subject. These are all qualities that are expected of a camera operator on a film set. Real world filming is, however, burdened with the use of special wheeled structures for moving cameras smoothly: cranes and purpose-built scaffolding are needed for elevated camera placement; special camera housing must be used for under water shots and so on (Brown 2002). With bots, these problems do not exist. Bots can move smoothly or in a jerky fashion as required; a bot can be any height, easily achieving so-called doggie-cam shots or crane shots; bots can fly and swim and so provide footage from otherwise involved angles.

We have created a number of these `CameraBots`, as we call them, for use in shooter games and each one acts autonomously. Each one is named according to the situation in which it is used or for the cinematic shot that it carries out (see Section 2.2 for the basis of these):

- *`EstablishCam`*: The establishing shot `CameraBot` establishes rooms with a long shot.
- *`CharacterCam`*: The character shot `CameraBot` introduces characters with full shots.

- *NaviCam*: The navigation `CameraBot` films the game world from behind the player's avatar and moves with the avatar to allow the player to navigate. The view provided depicts the avatar at the bottom of the screen and the rest of the scene in the direction in which the avatar is looking.
- *CombatCam*: The combat `CameraBot` is designed for combat situations, i.e. shooting at enemies. It provides a first-person perspective.
- *MissileCam*: The missile `CameraBot` follows a missile fired by the player if well aimed, and films its effect, i.e. the enemy taking fire.

	Perspective	**Orientation tied to user input**	**Subject**
EstablishCam	Third-person	No	No
CharacterCam	Third-person	No	Yes
NaviCam	Third-person	Yes	Yes
CombatCam	First-person	Yes	No
MissileCam	Third-person	No	No

Table 4.1: CameraBot classification

Each of the five `CameraBots` we have created can be classified according to whether it films in first- or third-person perspective, whether its orientation is tied to user input (usually the movement of the mouse) or not and whether it films a single subject or not (`CombatCam` is not seen to film a subject but rather from a subject's perspective). Some `CameraBots` update their placements continually while others do so at the command of the `Cinematographer` depending on the classification. See Table 4.1.

We will now treat each `CameraBot` type in turn and highlight the difference between the way each one places itself (sets its position and orientation) in the game and determines whether its current shot is valid or not. The significance of shot validity is that the `Cinematographer` uses it to aid in editing decisions (if the active `CameraBot`'s shot is invalid the `Cinematographer` will cut to another). The `CameraBots`, whose orientations are tied to user input, namely `NaviCam` and `CombatCam`, update their placement on a continual basis, i.e. they do so independently of the `Cinematographer` and whether they are providing the player view or not. On the other hand, the orientations of `EstablishCam`, `CharacterCam` and `MissileCam`, are not tied to user input and these only update their positions when issued a reposition command by the `Cinematographer`.

4.2.1 NaviCam

`NaviCam` finds a position behind the avatar's head and looks in the same direction as it (i.e. the direction set by the user input), such that a clear view of it and the game world is provided. See Figure 4.2.

Figure 4.2: `NaviCam`'s view: The avatar is positioned towards the bottom of the screen

If the shot does not fulfil certain criteria `NaviCam` sets it to invalid. Pseudo code for its update cycle is as follows:

```
NaviCam Placement
1. set shot to valid
2. set camera orientation to avatar orientation
3. set camera position to avatar's eye position
4. move camera back
5. if line of sight to avatar is obstructed move closer until clear
6. if camera is very close to avatar or if avatar is crouching
7.    set shot to invalid
8. end if
```

Line 5 of the pseudo code suggests a testing of the clarity of the line of sight from the avatar to the `CameraBot`. Every `CameraBot` can do this by tracing a box through space (Figure 4.3).

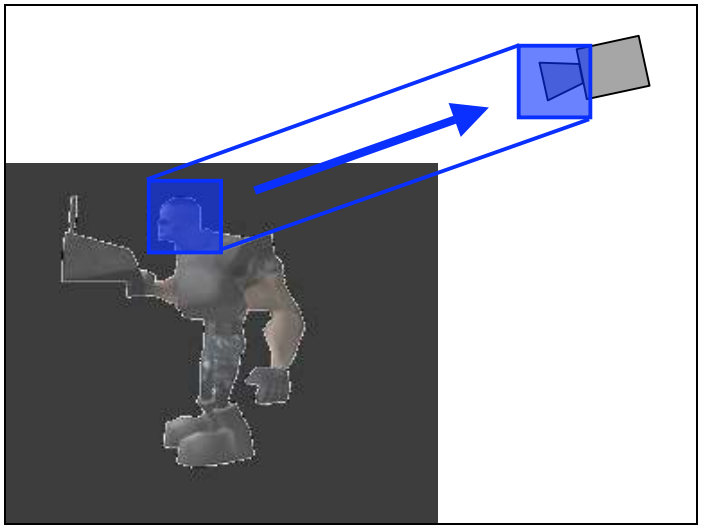

Figure 4.3: Testing for line of sight between avatar and `NaviCam`: A box is traced through space

In this case `NaviCam` traces a box from the avatar to its own position. Lines 6 to 8 set `NaviCam`'s shot to invalid if the avatar is either too close to the `CameraBot` or crouching. This is because, if the `CameraBot` and avatar become too closely positioned, as may happen as the avatar backs into a wall, the avatar will start to fill the view and the scene will become occluded. Also, if the avatar is crouching, it is likely that the player is moving it underneath a low ceiling, in which case `NaviCam` will not be able to provide a

view of both the avatar and the scene so it simply sets its shot to invalid until the situation changes.

4.2.2 CombatCam

Figure 4.4: `CombatCam`'s view

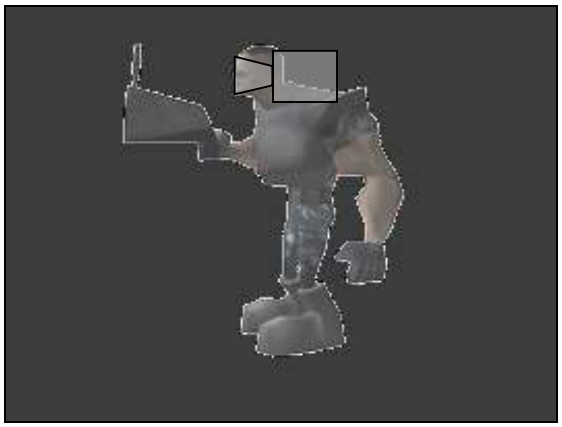

Figure 4.5: The positioning of `CombatCam`

The second `CameraBot` whose orientation is tied to user input, `CombatCam` (Figure 4.4, Figure 4.5), simply positions itself at the avatar's eyes and again looks in the same direction; this is the standard view used in FPS games. It never sets its current shot to invalid because the avatars eyes will always be at a valid location in the game. We chose it for combat situations as it gives the player more accuracy when shooting enemies. Its update cycle is below:

```
CombatCam Placement
1. set camera orientation to avatar orientation
2. set camera position to avatar's eye position
```

4.2.3 EstablishCam

`EstablishCam`, `CharacterCam` and `MissileCam`, i.e. the `CameraBots` whose orientation is not tied to user input, only update their positions when issued a reposition command by the `Cinematographer` and not during every cycle of the game. Since in cinematography there is a limit to how often one should employ an establishing shot we apply time restrictions to ensure that establishing shots and also missile shots do not occur too frequently. We now consider the positioning of these three `CameraBots`.

Figure 4.6: `EstablishCam`'s view

`EstablishCam` films a static long shot of the room the avatar occupies if one has not already been filmed in this vicinity.

```
EstablishCam Placement
1. if we have already filmed an establishing shot close to here
2.    shot is invalid
3.    return
4. end if
5. if we haven't filmed an establishing shot recently
6.    find a long shot of the room that includes the avatar
7.    if none found
8.          shot is invalid
9.    else
10.         shot is valid
11.         set last establishing shot time
12.         record establishing shot position
13.   end if/else
14.else
15.   shot is invalid
16.end if/else
```

Lines 1 to 4 ensure that multiple establishing shots are not filmed in the same locality. Line 5 applies the time restriction referred to above. Line 6 finds the shot and Lines 7 to 14 carry out operations depending on the validity of the shot. Specifically, if the shot is valid, the timing and placement of the shot are recorded; otherwise no action is taken and the `Cinematographer` will see that the shot is not valid.

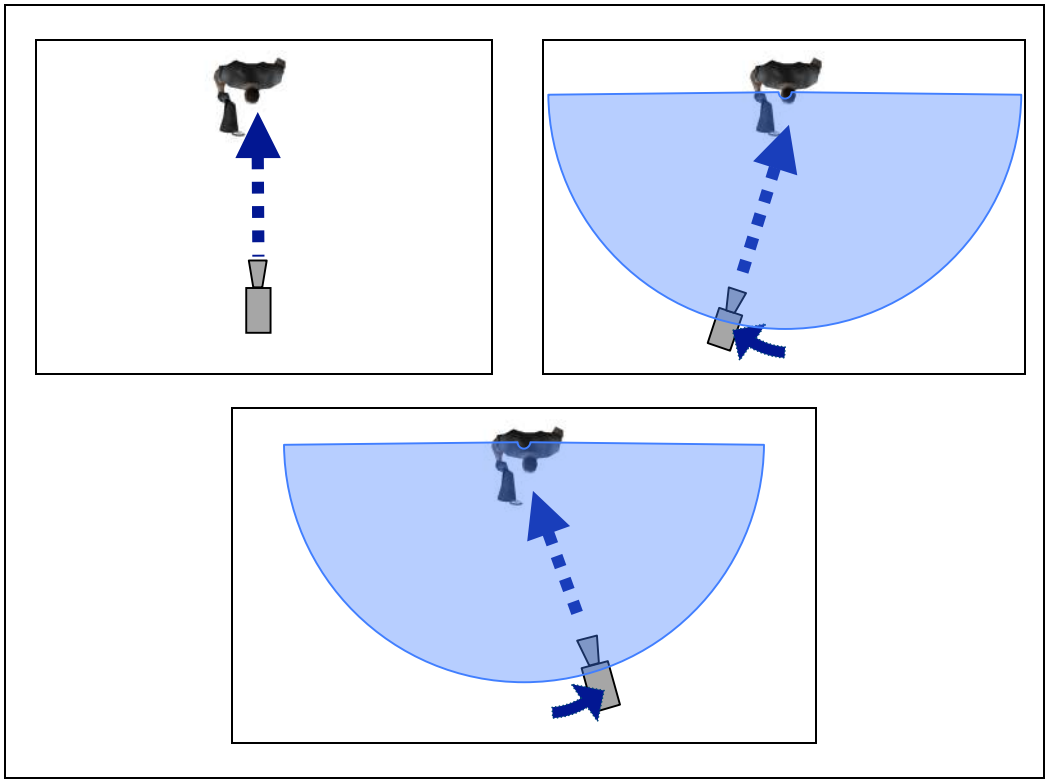

Figure 4.7: `EstablishCam` begins with a frontal view of the avatar. If this is not clear it tries increasingly large deviations from this angle.

The method that `EstablishCam` uses to find a suitable shot is to begin with a shot a specified height above the avatar and facing the front of the avatar (Figure 4.7). If the view of the avatar is not clear `EstablishCam` tries increasingly greater subject angles on alternate sides. If one is not found and `EstablishCam` has traced out a full 180º it sets its shot to invalid.

4.2.4 CharacterCam

`CharacterCam` films static full shots of characters occupying the same room as the avatar. The characters filmed are always enemies so it uses a three-quarter low angle to

evoke a sense of apprehension. If the full shot is not clear, it moves onto the next subject in the room. It ensures that each subject is established, i.e. has enough screen time, before moving on to the next.

```
CharacterCam Placement
1. while we have not filmed the last subject in the room and the current
   shot is either invalid or complete
2.    film a full shot of the next subject
3.    if a clear view of the subject
4.          shot is valid
5.    else
6.          shot is invalid
7.    end if/else
8.
9.    if this shot is valid
10.         record the start of the shot
11.   end if
12. end while
```

The first line of the pseudo code provides for the entering of a decision loop in the event that the current shot is either invalid or complete. The remaining lines test for clear shots of the remaining subjects in the room and set the shot to valid if one is found. These line of sight tests are again performed by tracing a box through space.

4.2.5 MissileCam

`MissileCam` (Figure 4.8) films a static shot of a missile and its target. In this case line of sight tests are performed for two subjects.

Figure 4.8: `MissileCam`'s view: The trail of the missile can be seen moving towards its target

See pseudo code below.

```
MissileCam Placement
1. if we have not filmed a missile shot recently
2.     find a long shot of the missile and target supplied (by the
       Cinematographer) that includes both, is aimed at their mid-point,
       and is at right angles to the travelling direction of the missile
3.     if none found
4.           shot is invalid
5.     else
6.           shot is valid
7.           record last missile shot time
8.     end if/else
9. else
10.    shot is invalid
11.end if/else
```

`MissileCam`'s placement method is similar to that of `EstablishCam` except that it has two subjects – the missile and its target – and it aims at their mid-point. It films from

either side of the action axis as defined by the missile's direction of motion and uses a 60º arc on each side to find a suitable placement. See Figure 4.9 below. As with all other `CameraBots` it traces a box from both subjects to its own placement to test for a clear view. Again, if none is found it sets its shot to invalid.

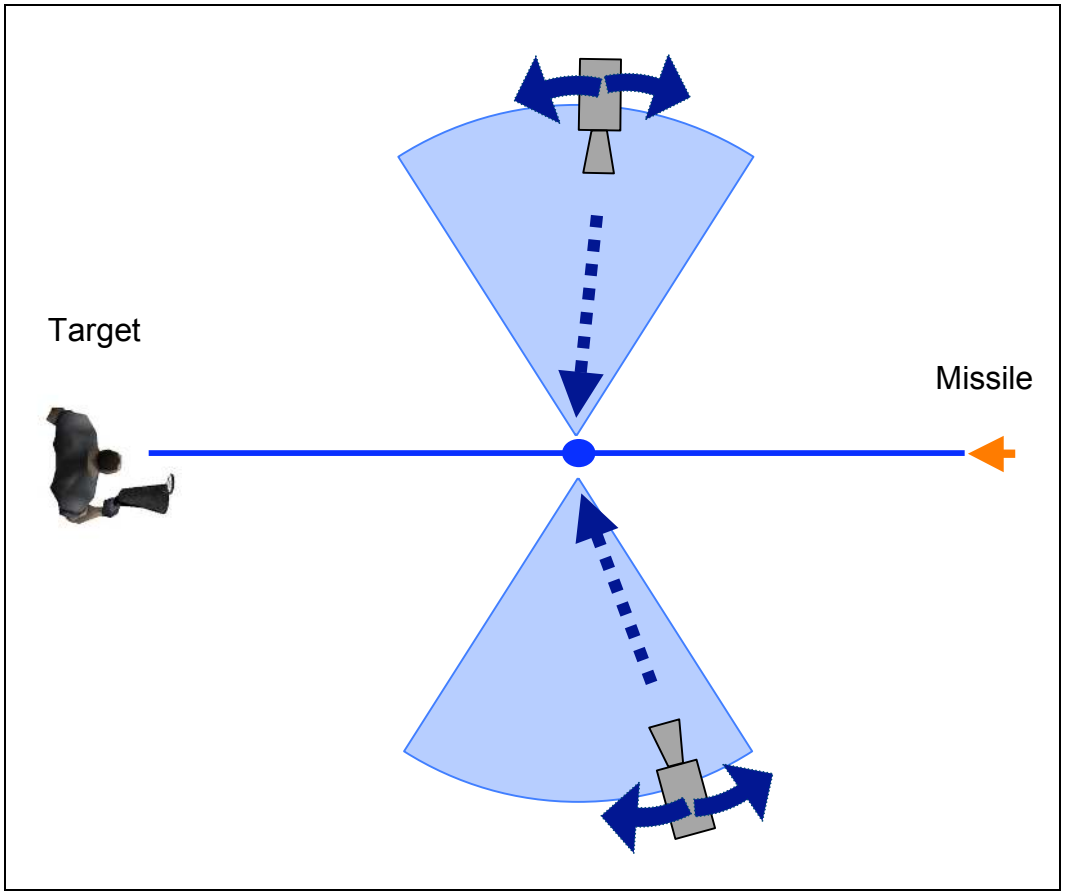

Figure 4.9: The positioning of `MissileCam`: The relative distance from `MissileCam` to the mid-point between subjects is reduced for illustrative purposes

The result of each update cycle or positioning function discussed above is that a given `CameraBot` has found a valid shot or not and sets the shot to valid or invalid accordingly. In addition, each `CameraBot` sets its shot to be complete or incomplete depending on certain factors. `CharacterCam`'s shot is complete if every subject in the room has been given enough screen time; every other `CameraBot`'s shot is complete if it has been active long enough. Now the `Cinematographer` can use this setting in

combination with the validity of each CameraBot's shot in order to decide which one to set as the player's view.

4.3 Cinematographer

The Cinematographer is a software entity with no representation in the game. It exists solely in our virtual cinematography system unlike the CameraBots which, although invisible, take up a location in the game world. The job of the Cinematographer is to create new CameraBots; have them placed in the game world, coordinate them, and select the appropriate one to provide the player's view at each instant. At the beginning of a game the Cinematographer creates one of each of the five types of CameraBots discussed above (EstablishCam, CharacterCam, NaviCam, CombatCam and MissileCam), has these placed in the game world (this is discussed in detail in the following chapter) and sets NaviCam to be the player's view. During each time cycle, the Cinematographer assesses the state of the game to decide whether to cut to, and perhaps reposition, a new CameraBot or to continue using the current one. As already mentioned, it uses the introductions principle from cinematography (see Section 2.7) to inform this decision. Making introductions in a shooter game, however, requires the use of extra cinematic devices because of the nature of the game.

In particular, during the game-play within FPS games, and many other genres, the player controls his avatar at all times. This constant association with the avatar poses an obstacle to extending the number of shots provided, since for example, it doesn't make sense for CharacterCam to show the player a shot of another character if he is still controlling the avatar off-screen. Moreover, this particular character might even be shooting the avatar at the time and now the player cannot avoid the attack. Our solution to this problem is to have the Cinematographer freeze the current frame in order to introduce the new setting and new characters, i.e. when EstablishCam or CharacterCam are active. When the introductions have been made, the game resumes. This feature means that our system can only be applied to single player games because it

would be impractical to freeze an entire multiplayer game for one player's establishing shot.

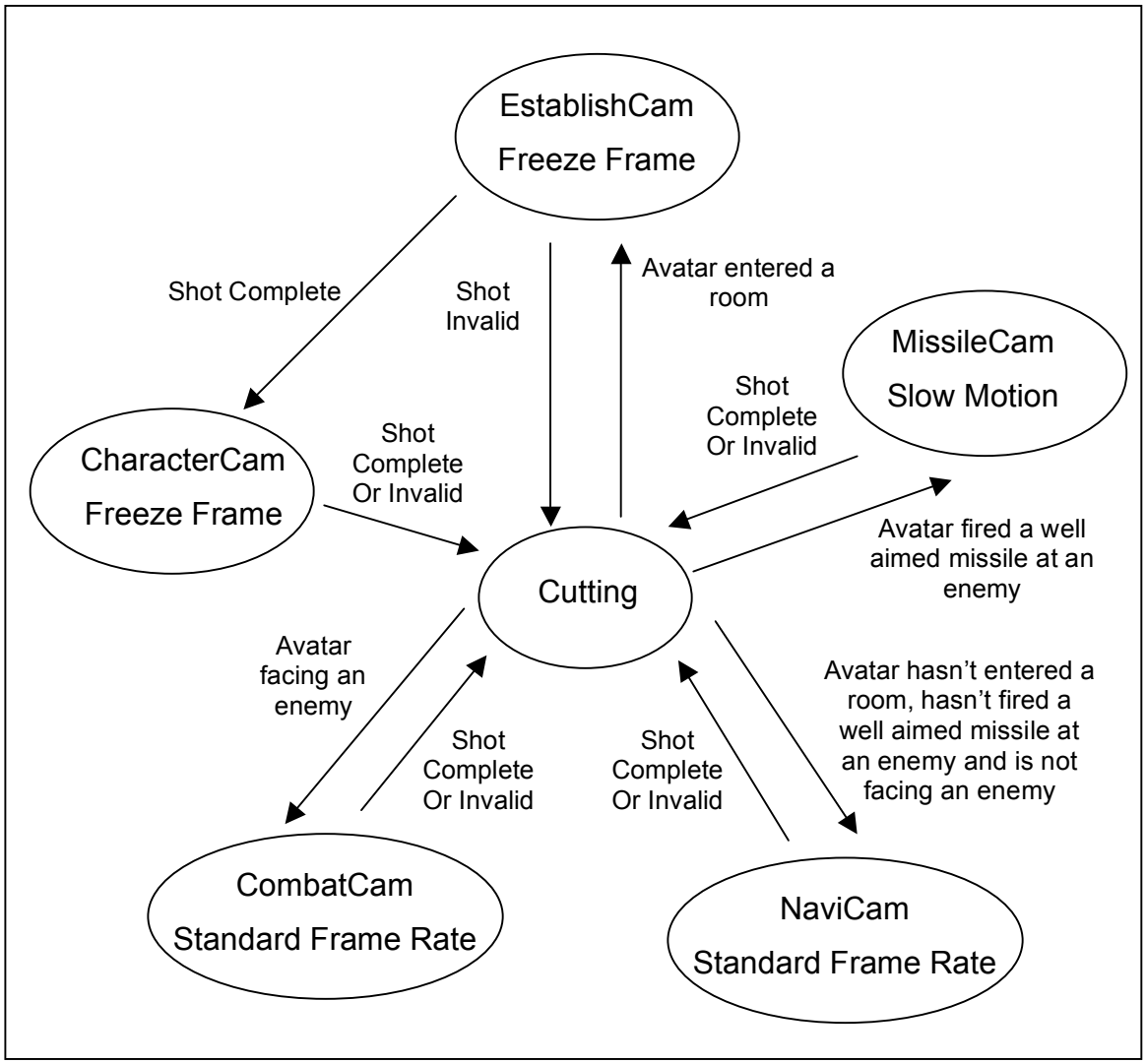

Figure 4.10: `Cinematographer` Finite State Machine: This outlines the editing decisions made by the `Cinematographer`. As can be seen above when most `CameraBots` have a complete or invalid shot the FSM moves to a "Cutting" state. This represents the intermediate state between one `CameraBot` and the next. These editing decisions are covered in more detail in the pseudocode below.

Specifically, in accordance with the introductions principle, if the avatar enters a room it has not occupied before, the `Cinematographer` cuts to `EstablishCam` and turns on freeze frame (Figure 4.10); when the room is established it cuts to `CharacterCam`;

when the characters are established it turns off freeze frame and cuts to one of the other three `CameraBots`. During normal game-play the `Cinematographer` usually activates `NaviCam` unless the avatar is facing an enemy in which case it cuts to `CombatCam`. If the avatar fires at an enemy with a projectile weapon (a blaster bolt or a rocket, as opposed to a *hitscan* weapon, which does not fire projectiles but simply hits anything in its path instantaneously), the `Cinematographer` applies a special shot to this action by turning on slow motion and cutting to `MissileCam`. Now the player can watch the success or failure of his well-aimed shot in slow motion. The use of slow motion further constrained the application of our system to single player games. The pseudo code for the `Cinematographer`'s finite state machine which both represents the state of the game and includes directives for the coordination of `CameraBots` and editing decisions (see Section 2.6) is below. An explanation follows.

```
Cinematographer Update Cycle
1.    if EstablishCam is the player's view and has finished
      establishing this room
2.          reposition CharacterCam
3.          if CharacterCam has a valid shot
4.                cut to CharacterCam
5.                exit decision loop
6.          end if
7.          cut to NaviCam (because CharacterCam must not have found a
            valid shot)
8.          turn on standard frame rate to cancel freeze frame (see
            line 15)
9.    end if


10.   if EstablishCam is not the player's view and the current shot is
      complete
11.   or the current shot is not valid
12.         turn on standard frame rate to cancel possible freeze frame


13.         if avatar entered a new room
14.               reposition EstablishCam to establish the room
15.               if EstablishCam has a valid shot
16.                     turn on freeze frame
```

```
17.                     cut to EstablishCam
18.                     exit decision loop
19.               end if
20.         end if

21.         if avatar has fired a missile at an enemy and it is well
            targeted
22.               inform MissileCam of the missile and the enemy and
                  reposition MissileCam
23.               if MissileCam has a valid shot
24.                     turn on slow motion
25.                     cut to MissileCam
26.                     exit decision loop
27.               end if
28.         end if

29.         if CombatCam is not the player's view and avatar is facing
            an enemy
30.               cut to CombatCam
12.
31.         else if neither CombatCam nor NaviCam are the player view
32.         or
33.         CombatCam is the player's view and no enemy has been in
            range for a while and avatar is not attacking
34.               cut to NaviCam
35.               if NaviCam doesn't have a valid shot
36.                     cut to CombatCam
37.               end if

38.         else if NaviCam is the player's view and doesn't have a
            valid shot
39.               cut to CombatCam
40.         end if/else
41.   end if
```

We begin explaining the pseudo code from line 10 and will return to line 1 afterwards. Lines 10 and 11 test whether the current shot is complete or invalid. If so `EstablishCam`, `MissileCam`, `NaviCam` or `CombatCam` will be set as the player's view as the state of the game dictates (lines 12 to 40). Line 13 features a call to a room

test function. Room testing is explained in detail in the following section. The cutting of the view to `CombatCam` (lines 29 to 40) requires some further explanation. `CombatCam` will be used from the time that the avatar is facing an enemy until there have been no enemies in range for a while and the avatar is not attacking. If the `Cinematographer` were to test that the enemy were in the same room as the avatar rather than that the avatar were facing an enemy, the player would know any time an enemy was close but out of view because `CombatCam` would be turned on. This would more than likely detract from the game play so we chose to detect for the avatar facing an enemy instead. Generally, the FSM is designed such that if no other `CameraBot` can find a valid shot, `CombatCam` will be used. For example, if `NaviCam` is trying to shoot but the avatar is crouching, `CombatCam` is used until the avatar finishes crouching, perhaps while passing under a low ceiling. Lines 1 to 9 of the pseudo code cater for the situation where `EstablishCam` has been active but has now completed its shot. Here `CharacterCam` is turned on and instructed to find a valid shot, and if this is not possible `NaviCam` is used instead.

In addition to the types of shots, cutting principles and guidelines for shooting scenes used we consider continuity (Section 2.5) in our virtual cinematography system. Continuity is always preserved with a cut from `NaviCam` or `CombatCam` to any other `CameraBot` or from any other `CameraBot` to `NaviCam` or `CombatCam`. This is because when the avatar is moving, the action axis is drawn in the direction of its movement and when it is stationary the axis is drawn in the direction in which it is looking. `NaviCam` and `CombatCam` are always filming in the direction in which the avatar is moving or looking and therefore are shooting on the neutral axis. When a camera is filming on the neutral axis (see Chapter 2) and cuts to either side of the axis or is filming from either side of the axis and cuts to the neutral axis continuity is maintained. Unfortunately, there was not sufficient time to implement consistent screen direction for cuts from `EstablishCam` to `CharacterCam` and for `CharacterCam`'s cuts from subject to subject.

4.4 Room Detection

As mentioned above, the `Cinematographer` will try to turn on `EstablishCam` if the avatar has just entered a room. It is a separate software module called *Rooms* that determines whether or not the avatar is in a room at a given instant and also when this status has changed. The Quake II levels that we worked on for this entire project consist of rooms connected to each other by corridors. If the avatar is in a corridor or door frame, it cannot be in a room so we chose to test for this and return the inverse of the result. The room detection control structure is outlined below. Illustrations and explanations of each line follow.

```
Room Detection Algorithm
1. if the ceiling above the avatar is low
2.    and there are walls on opposite sides of and close to the avatar
3.    and there is a clear path for some distance perpendicular to the
      direction of these walls relative to the avatar
4.          return that the avatar is not in a room
5. end if


6. return that the avatar is in a room
```

Figure 4.11: Ceiling test: A trace for clearance is performed from below to above the avatar. The avatar is visible towards the bottom of the image

The ceiling test (line 1 of the above pseudo code) is performed with a single trace for clearance from floor to ceiling; if this distance is below a defined threshold, the ceiling is low at the avatar's current position (Figure 4.11).

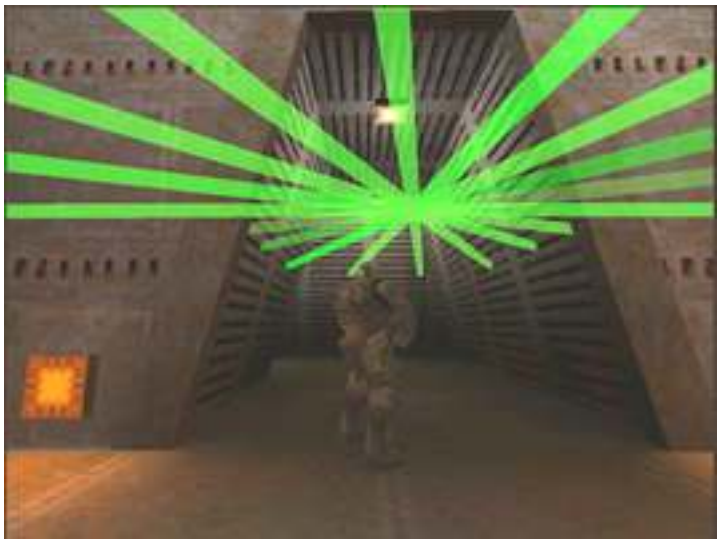

Figure 4.12: Opposing walls test: A number of clearance traces are performed in the horizontal plane.

The opposing walls test (line 2) is performed by firstly calculating the mid-point between floor and ceiling at the avatar's position; then testing for clearance from this position for a given distance in a given direction and the same distance but in the opposite direction in the horizontal plane; and performing this test from this point at nine angular intervals around the vertical (Figure 4.12).

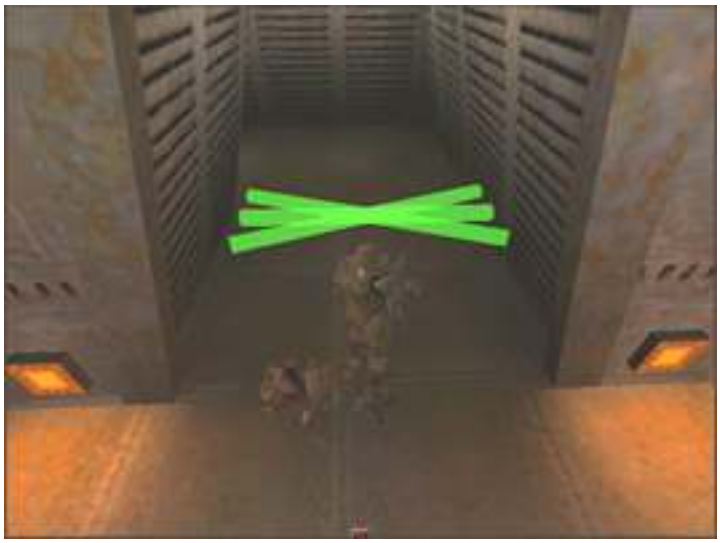

Figure 4.13: Multiple close opposing wall point pairs (marked by the lines between the pairs of points). In this case the pair represented by the middle line would be used for the clear path test

If any of these tests is unclear, there are opposing walls at the points where the traces collide with scene geometry. If one or a number of these pairs of points are found, the avatar might be in a corridor (Figure 4.13). If a number of pairs have been found, it is those that are closest to each other that will be used for the clear path test (line 3) because the line drawn between this pair of points is the closest to a perpendicular to the direction of the corridor.

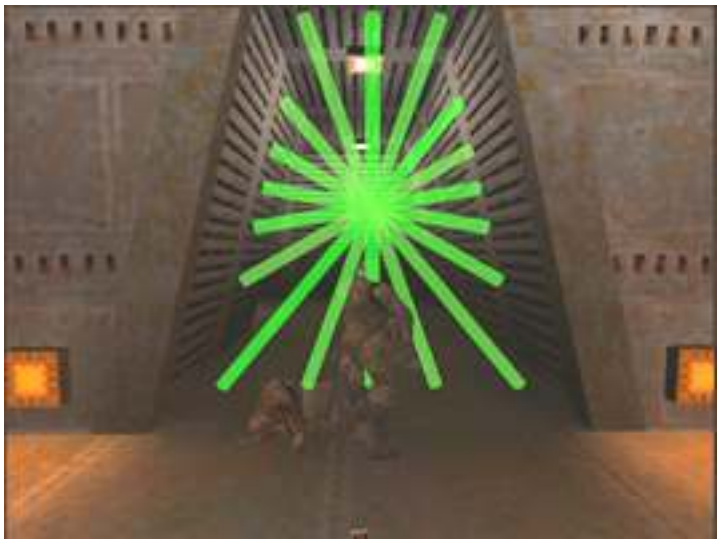

Figure 4.14: Further testing with radial traces

Before the clear path test, more wall, floor and ceiling detection is carried out to ensure that the avatar is indeed surrounded by geometry. A number of traces are performed in the vertical plane shared by the floor point and ceiling point found in the initial step (line 1), and the two chosen opposing wall points (Figure 4.14). If all of these traces are unclear, the avatar is surrounded by geometry; otherwise, further testing is aborted.

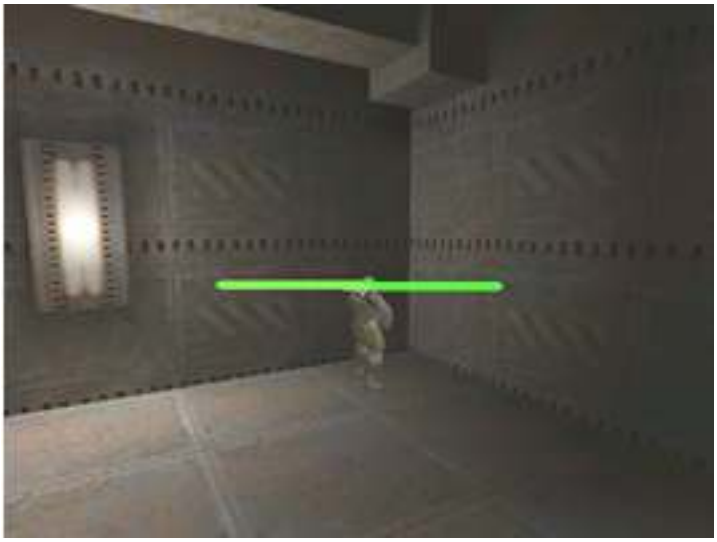

Figure 4.15: Corners at low ceilings posed a problem for room detection before the clear path test was added. Here the walls at this corner had been detected as opposing walls

Finally a clearance trace is performed to determine whether there is a clear path on both sides of this vertical plane. This test is required because if the avatar is standing in a corner of a room where the ceiling is particularly low the previous tests would detect it to be in a door frame or corridor (Figure 4.15).

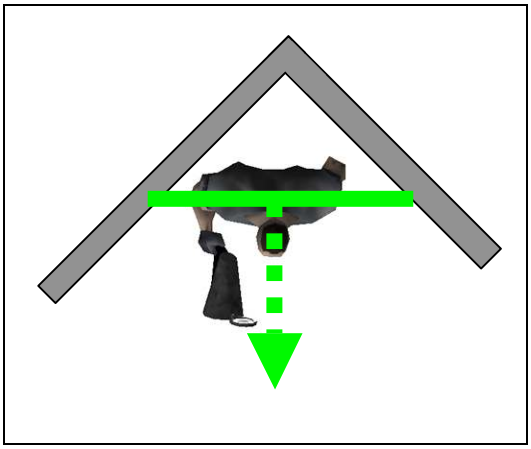

Figure 4.16: Avatar in a corner: Clearance in only one direction perpendicular to the opposing walls line

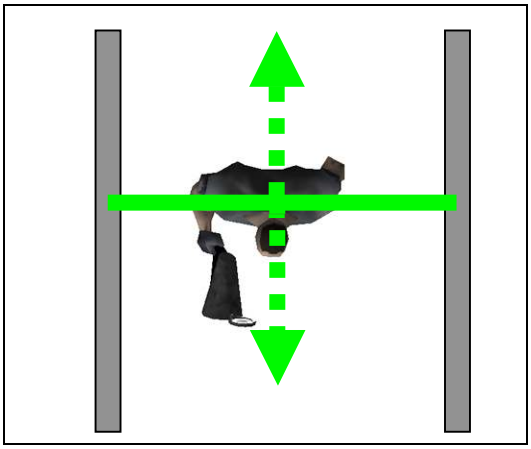

Figure 4.17: Avatar in a corridor: Clearance in both directions perpendicular to the opposing walls line

The difference between a corner, and a corridor or door frame is that a corner has a clear path perpendicular to the line defined by the opposing wall points in one direction only whereas corridors and door frames have both perpendicular clear paths (Figure 4.16 and Figure 4.17). The clear path test is illustrated in Figure 4.18. To summarise room detection, if the avatar is under a low ceiling, in-between two close walls and there is a clear path roughly parallel to these walls for a certain distance in both directions, it is not in a room. Thus, the room detection function returns the opposite of this.

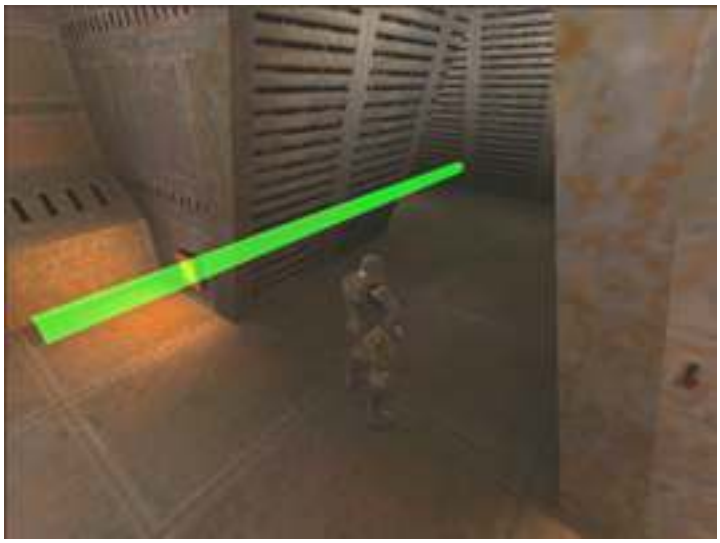

Figure 4.18: Clear path detection

An alternative for finding the most directly opposite wall points (line 2) might have been to take the wall normals, i.e. the angles of the walls, into account. While this would certainly work in most situations, it would not work for jagged walls so we rejected the idea. This concludes our discussion of the virtual cinematography system. We will now present the results from our testing phase.

4.5 Results

In order to test the suitability of our system to shooter games we had a number of participants play both our modified version of Quake II and the standard version to compare differences between the two in terms of each participant's success in the game. We used three different game levels and made sure that the combinations of levels and versions were evenly distributed. Of a total of 17 participants we tested:

1. How our system affected their ability to frag (kill) enemies in the game. One of the chief measures of success in Quake II is how many enemies you frag so we put this measure in the form of average number of frags per hour based on a playing time of up to nine minutes on each version of the game.

2. How our system affected their ability to collect items in the game. Another important task in the FPS genre is collecting items such as health, weapons,

ammunition, and play enhancers, such as *quad damage* which quadruples the amount of damage that a player inflicts on the enemies in the game. Again we calculated this per unit time as pick-ups per hour.

The results are as follows:

1. The ratio of frags per hour in our version of Quake II to that in the standard version is 1.26. This means that the participants, on average, fragged 26% more enemies per unit time while playing our version.

2. The ratio of pick-ups per hour in our version to that in the standard version is 0.89. This means the participants collected, on average, 11% less items per unit time with our version.

In addition to analysing the participants' game-play, we asked them to fill out a short questionnaire, the results of which are in Table 4.2 below. Some of these questions overlap with the results above but it is interesting to consider the participant's response because of the subjective nature of a visual experience such as this.

Question	**Average**
How did the camera work in the cinematic mode affect your ability to move around the game? 1: Made it easier, 5: Made it more difficult	3.21
How did the camera work in the cinematic mode affect your ability to find your bearings? 1: Made it easier, 5: Made it more difficult	2.95
How did the camera work in the cinematic mode affect your ability to find important items – health, ammunition etc.? 1: Made it easier, 5: Made it more difficult	2.21

How did you find the switch/transition from one camera angle to the next? 1: Natural, 5: Confusing	3.00
How did the camera work in the cinematic mode affect the overall game-play? 1: Made it more interesting, 5: Made it less interesting	2.26
How did you find the overall experience of playing in cinematic mode? 1: More like watching a film, 5: No different to standard mode	2.16
Do you think more games should have cinematic cameras like this one? +1: Yes, -1: No	0.58

Table 4.2: Results of questionnaire

Here we present an analysis of these questionnaire results. On average, players found that the cinematic mode made it a little more difficult to manoeuvre around the game. It allowed them to find their bearings, i.e. to maintain orientation, a little bit better. It enabled them to find pick-ups more easily. Players found cuts to be mid-way between natural and confusing. They found the game-play more interesting in cinematic mode and thought it was a bit more like watching a film than playing a game. Most of the participants thought more games should have a virtual cinematography system like ours.

Chapter 5

Implementation

In Chapter 4 we discussed the operation of the different components of our virtual cinematography system. In this chapter we consider the specifics of their implementation. We implemented the system as a modification to the Quake II game engine (id Software 1997) and here we present the idea of the game engine in general, our modifications to the Quake II game engine, the interface between our cinematography system and Quake II, the hierarchical structure of the `CameraBots`, and difficulties encountered during the project.

5.1 Quake II

A game engine is a computer program that provides the time critical functionality of a game such as rendering, networking, and audio processing but also other commonly used functionality such as low level AI for the bots (Sánchez-Crespo Dalmau 2004). The advent of the game engine meant that the game logic (the rules of the game), game levels (these are called maps), character models, and artwork could be created separately so that people working on different parts of a game could do so independently to a certain degree. The game engine that we have chosen to work on is Quake II. The game (id Software 1997) was released on November 30 1997. The full source code was then released under the terms of the Gnu Public License (Gnu) on December 21, 2001. Quake II is written in the C programming language and uses a client-server topology which allows for games to be played over a computer network. A server runs the game and one or a number of clients connect players to each other via the server. As each player on each client interacts with the game, messages are sent to the server to reflect these

proposed changes to the game world (i.e. the moving of the avatar and so on). The server considers all messages from all clients, resolves the game world accordingly and sends the updated game state to each client. Specifically the server consists of part of the game engine which drives the game and the *game DLL* which contains code specific to the particular game including its rules. The other part of the game engine is the client. Each client computer must have a game engine but not necessarily a game DLL but the server has both. This means that any client computer which has the game DLL can host a multiplayer networked game and will therefore also be a server. For single player games Quake II still operates with a client-server topology, the two simply reside on the same computer. It is single player games for which we have developed our virtual cinematography system for the reasons discussed in Section 4.3.

A number of modifications (*mods*) have been made to the Quake II game DLL by the game player community since its release giving rise to improvements to different aspects of the game (such as improved lighting, better weapons and so on) and even entirely different games altogether. These include a C++ version of the game DLL which allows "modders" to implement their modifications with some object-orientation. For our work we modified the C++ game DLL and also made some small modifications to the client part of the engine itself (in C) to give us full control of the viewing system. We used the C++ version of the game DLL so that we could implement an object-oriented representation of the `Cinematographer` and the `CameraBots`. Our `Cinematographer` and `CameraBots` are domain independent: The code for these objects communicates with the underlying game system via two other objects that we developed called the game-cinematography interface (GCI) and the Avatar object. This means that our cinematography system can be adapted for use with other game engines by rewriting of some, but not all, of the member functions in the GCI and Avatar.

In order to fully explain how the GCI and Avatar plug into Quake II it is first necessary to explain the different cycles that occur to keep the game running. The client-server topology means that Quake II has two cycles – the server cycle which applies to all clients and the client cycle specific to each. Every 100 milliseconds the server frame occurs on the host computer where the server runs some networking and commands the

game DLL to update the game world. The game DLL runs the bots, updates all of the entities in the game world including players' avatars, and so on. The client frame occurs more frequently on each player's computer. This is where input from the player is taken and some client-side prediction is performed so that an updated rendering can be provided between server frames. Client side prediction was especially important when the game was released because a considerable number of players would have been using a slow network connection in multiplayer mode meaning that server updates would have been relatively infrequent. It can only, however, take user input into account when creating a view. The client sends its predicted state of the avatar to the server during the server frame; the server merges this with the state of the rest of the game and other clients' predictions; and returns the updated game state.

5.2 Quake II Modifications and the Game-Cinematography Interface

It was decided early in the development of our cinematography system to keep the cinematography code independent of the game code. This required the creation of an interface class with a number of functions that allow the cinematography code to use the game code without knowing its underlying structure. A principal motivation for creating this game-cinematography interface, or GCI, was that it makes the cinematography code portable, i.e. it can be used in another game engine with only small modifications to the GCI. Later on it was found that a good proportion of the GCI functions were related to the avatar and so an interface to the avatar was created in the form of the Avatar class, and these functions moved to it. The Avatar class will be discussed in the following section. Another advantage to the GCI is that it simplified the cinematography code a great deal by bringing only the relevant aspects of the game code to the fore, and allowed us to use function names that are consistent with those in the cinematography code. In this section we discuss the GCI and the modifications that were made to the Quake II client and game DLL.

It is during the server frame that our cinematography code is updated. When the server requests the game DLL to update the game, the game DLL requests the GCI to update

also. The GCI, in turn, calls update functions on the Rooms object (see Section 4.4), the Avatar and the `Cinematographer` (where the FSM is updated – see Section 4.3). It also updates the avatar's 3D model if third-person mode is on (also to be discussed shortly). We needed the GCI to perform various tasks for the cinematography code including:

- To accept any `CameraBot` as the player's view
- To switch the viewing system from first-person to third-person for certain `CameraBot` types (see Section 4.2)
- To test for clear paths in the game environment.

In addition we needed the GCI to facilitate some cinematic devices including:

- Freeze frame
- Slow motion
- Widescreen (Quake II is displayed at an aspect ratio of 4:3)

As a solution to the first requirement we provided a function that allows each `CameraBot` to request that it become the player view directly, given an instruction by the `Cinematographer` to do so. On this request, the GCI simply takes note of the active `CameraBot`. During every server frame the game DLL queries the GCI for the active `CameraBot` and applies it to the player's view, so any change will be applied in the following cycle.

When third-person `CameraBots`, i.e. `EstablishCam`, `CharacterCam`, `NaviCam` and `MissileCam`, are instructed by the `Cinematographer` to become active, they instruct the GCI to turn on third-person mode. This is necessary because usually the first-person view that Quake II provides includes the avatar's hand and weapon (Figure 5.1) but the third-person perspective should show a full 3D model of the avatar (Figure 5.2). Therefore, on request, the GCI turns off the hand 3D model and creates a new entity in Quake II to represent the avatar model. The actual avatar entity remains invisible as in first-person mode, so that enemies will attack the newly created entity instead. We had to

create this new entity because a number of functions in Quake II expect the player's view and the avatar's representation in the game to be at the same location, and will not operate correctly otherwise. We also had to modify the way Quake II changes the colour tone of the display when the avatar is under water so that this occurs, instead, when the `CameraBot` providing the player's view is under water.

Figure 5.1: The first-person perspective view used in Quake II

Figure 5.2: Third-person perspective used by `NaviCam`

Another requirement for our cinematography system was that the orientation of `CombatCam` and `NaviCam` should be tied to user input (see Section 4.2) whereas that of `EstablishCam`, `CharacterCam` and `MissileCam` should remain independent. A function called `setFixedView` allows each `CameraBot` to request that the GCI make the appropriate setting. The game DLL then sends this request to the client which either ties the player view to user input or not as per the requirement but also puts the player's view in wide screen for the input independent views. This is an important feature because the input independent views are used in conjunction with either freeze frame (for `EstablishCam` and `CharacterCam`) or slow motion (for `MissileCam`) and we found that when developing these views initially, we needed to give the player some indication that he had temporarily lost control of the avatar and that the viewpoint had moved away from the avatar. The widescreen view makes both of these points quite obvious and is actually already used in some commercial games (Ubisoft 2003) to indicate a switch to a cut-scene.

As covered in the previous chapter, the `Cinematographer` requests that the game is frozen for an establishing shot (it remains frozen for any ensuing character shots also). The GCI registers this request and while the game DLL is running the server frame, if it sees that such a request is registered, it simply does not update the entities in the game. This means that the avatar and its enemies remain frozen in position until the end of the establishing shot or character shots that follow. Slow motion is achieved similarly: The cinematographer requests the GCI to establish slow motion; the GCI registers this and every second server frame sets the game to be frozen so that half of the frames play the action and half do not.

As seen in Section 4.2 in the discussion on the placement algorithms of the `CameraBots`, objects in the cinematography code need to test for clear paths from one point to another in the game world during a game. A suitable function called a trace function already exists in Quake II. This traces a cuboid through the 3D environment, tests for collisions and returns the relevant data such as the fraction of the trace that was completed before a collision occurred. The GCI simply passes requests onto this function and returns the results.

Table 5.1 below contains the entire set of functions provided by the GCI.

`getAv`	`Return a pointer to the Avatar object`
`update`	`This is called every server frame. Update Rooms, Avatar, Cinematographer and avatar's 3D model if in third-person mode.`
`setTPMode`	`Set third-person mode on or off`
`getTPMode`	`Return the current setting`
`setFixedView`	`Connect the view orientation to user input or remove the connection them`
`getFixedView`	`Return the current setting`
`freezeFrame`	`Turn on freeze frame`
`slowMo`	`Turn on slow motion`
`action`	`Turn on normal camera speed`
`updateDisplayEnt`	`Update the third-person avatar model`
`traceVolume`	`Trace a path through the 3D world`
`traceAvatarToBot`	`Trace a path from the avatar to a bot`

time	Return the current time
placement	Return the placement (both position and orientation) of an entity
pos	Return the position of an entity
ori	Return the orientation of an entity
eyePos	Return the position of an entity's eyes
BBRadius	Return an entity's bounding box radius, i.e. that of a sphere approximate to the bounding box. Bounding boxes are cuboids that enclose an entity and are used for collision detection among other things.
resetEnemies	Reset the counter for cycling through the avatar's enemies (see nextEnemy and lastEnemy)
setEnemies	Set the counter to a certain index (entities are referenced by indices in Quake II)
getEnemies	Return the counter's current setting
nextEnemy	Return the index of the next enemy from the counter's current setting
lastEnemy	Returns true if this is the last enemy in the world, i.e. there are no enemies with an index higher than this one
newBot	Create a new CameraBot in the game
updateBot	Called by the game to update individual bots
placeBot	The CameraBots use this to position and orient themselves in the game
getBotPlacement	Return the placement in the game of a CameraBot
setBotBoundingBox	Set the bounding box of the bot for collision detection
setView	Register a CameraBot to be the player's view
getView	Return the CameraBot that is currently the player's view
clearBots	Use by the GCI on shutdown to remove the CameraBots from the game

Table 5.1: The member functions of the GCI (Game-Cinematography Interface)

5.3 Avatar

The avatar, as mentioned, abstracts the cinematography code from the game code but specifically from game code particular to the avatar. As can be seen from Table 5.2 below, a number of the member functions simply return attributes of the avatar such as position, velocity and so on. This might seem trivial but, again, it means the

cinematography code does not need to know specifics about the game code which in turn means the cinematography code can be applied to other game engines by rewriting portions of the Avatar and the GCI. The `facingEnemy` and `enemyInRange` member functions are both used by the `Cinematographer` to decide whether to turn `CombatCam` on or off respectively (see Section 4.3 for details). `facingEnemy` tests if an enemy is in the avatar's field-of-view and `enemyInRange` tests if the avatar has a direct line-of-sight to any enemy. If the avatar is facing an enemy, the `Cinematographer` will cut to `CombatCam` and later, if no enemy is in range anymore, to another `CameraBot`.

The `resetEnemies`, `nextEnemyInRoom` and `lastEnemyInRoom` functions are all used by `CharacterCam` to find enemies to film in the room occupied by the avatar. `nextEnemyInRoom` simply returns an index to the next enemy in the room, which `CharacterCam` can use to query positional information from the GCI. If either `missileToEnemy or newRoom` return true, the `Cinematographer` has the option to cut to `MissileCam` or `EstablishCam` respectively. `missileToEnemy` means the avatar has fired a missile at an enemy and it is well aimed whereas `newRoom` means the avatar has entered a new room. This function uses the Rooms object (see Section 4.4) to determine whether the avatar has entered a new room or not and during these room detection tests the general direction of the room relative to the point at which the avatar entered it is recorded. `EstablishCam` queries this setting via the `roomDir` function to determine how to position itself correctly.

`update`	`Called by GCI every server frame`
`setStates`	`Called by GCI at the end of cinematography code update to set previous data for next frame such as previous position, whether the avatar was previously in a room or not and so on`
`index`	`Return the avatar's index in Quake II`
`placement`	`Return the avatar's placment`
`pos`	`Return position`
`ori`	`Return orientation`
`eyePos`	`Return eye position`

prevEyePos	Return eye position in the last server frame
velocity	Return velocity
attacking	Return true if the avatar is attacking
healthFraction	Return the fraction of total health the avatar has
freeze	Freeze the avatar's position (for freeze frame)
crouching	Return true if the avatar is crouching
enemyInRange	Used by enemyRecentlyInRange (see below)
enemyRecentlyInRange	Return true if the avatar has recently had a clear path to an enemy
resetEnemies	Reset the counter for following methods
nextEnemyInRoom	Return the next enemy in this room
lastEnemyInRoom	Return true if this is the last enemy in this room
facingEnemy	Return true if the avatar is facing an enemy
aim	The orientation of the avatar's weapon
missile	Return true if the avatar is firing a missile this frame
missileToEnemy	Return true if the avatar is firing a missile this frame and it's well targeted
inRoom	Return true if the avatar is in a room as opposed to a corridor or door frame
enteredRoom	Return true if the avatar was not in a room during the last frame but is this frame
exitedRoom	Return true if the avatar was in a room last frame but is not this frame
newRoom	Return true if the avatar entered a room it hasn't entered before
roomDir	Return the general direction of the room relative to the avatar

Table 5.2: The member functions of the Avatar object

5.4 CameraBots

In Section 4.2 we discussed the general operation of the `CameraBots`. We will now discuss their object orientation hierarchy. At the top of the hierarchy is the `CameraBot` object (Figure 5.3 below). Every `CameraBot` type derives from this. A noteworthy implication of this is that it is very easy to plug new types of `CameraBots` into the cinematography system and not only expand its use for shooter games but apply it to

other genres also. This `CameraBot` class is abstract, that is, a `CameraBot` object cannot be created – only one of a derived class – and it has some virtual functions which means that the `Cinematographer` can call these functions on each derived `CameraBot` object without knowing the specific type of `CameraBot` it is (`EstablishCam` and so on).

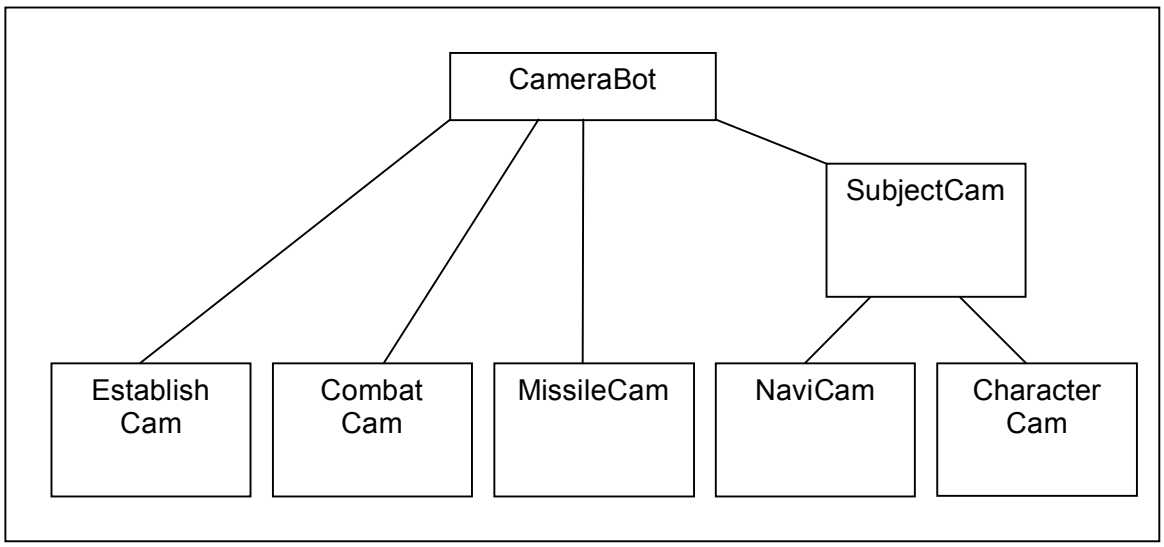

Figure 5.3: The `CameraBot` hierarchy

These virtual functions are: `update`, `setToView` and `shotComplete`. Each type of `CameraBot` does not necessarily have to perform any tasks with its derived version of the `update` function – `EstablishCam` and `MissileCam` do not – but `NaviCam` and `CombatCam` use it to maintain their positions relative to the avatar whether active or not, and `CharacterCam` uses it to continue looking for more enemies in the room to film but only when active. In an earlier version of the system we used the `CameraBot` class' own implementation of `update` to add Dutch tilt to the derived `CameraBot` when the avatar was dying. As discussed in Section 2.2, Dutch tilt is used in the cinematographer's vocabulary to imply that the subject is intoxicated, delirious or indeed dying. We found, however, that it was not quite as effective as hoped so it was removed. Every `CameraBot` must implement the `setToView` function because it is here that each one:

- Sets the game mode to either third- or first- person via the GCI (`CombatCam` is the only first-person `CameraBot`)
- Registers itself as the player's view
- Sets the orientation of the player's view to be tied to user input or not
- Records the time at which it has been activated

`shotComplete` is defined in the `CameraBot` class – it returns true if the requisite *shot time* particular to each `CameraBot` has elapsed since its activation – but a `CameraBot` can define its own version if necessary; `CharacterCam`'s shot is only complete if the current subject's shot is complete and it is the last subject to be filmed in this room. The one non-virtual function in the `CameraBot` class is `isShotValid`. `CameraBot` defines this to return the validity of the current shot as set by the derived class. As already seen, if the `Cinematographer` finds the active `CameraBot`'s current shot to be invalid it will cut to another.

5.4.1 SubjectCam

Two of our `CameraBots` (`NaviCam` and `CharacterCam`) are positioned a fixed distance from a subject in third-person perspective (as opposed to filming general shots of a scene like `EstablishCam` etc.) and since a number of shots in cinematography, and therefore possibly `CameraBots` that will be added in the future, also have this general configuration we decided to add another object in the hierarchy: the SubjectCam. The SubjectCam is another abstract class that derives from `CameraBot` and is, itself, derived from by `NaviCam` and `CharacterCam`. Since `MissileCam` technically has two subjects, the missile and its target, and `CombatCam`'s `update` code is so simple that using SubjectCam would add extra calculations, neither are child classes of SubjectCam. On creation, SubjectCam is given the relative position to the subject required and the index of the subject itself so that its placement can be queried of the GCI. Its `update` member function simply sets the `CameraBot`'s position based on the required relative position, tests for a clear view and moves the camera closer if occluded until it becomes clear. As mentioned, occlusion tests are performed by tracing a cuboid through space. An

obvious advantage to using SubjectCam is that it simplifies the derived class' code a good deal and many shots in cinematography, e.g. the full shot, medium shot, close-up and so on, are suited to its use.

5.5 Other Objects

In our virtual cinematography system we include some other objects that perform common tasks to enhance its portability. For example, numerous calculations regarding 3-dimensional vectors are performed by most modules in any 3D environment including our virtual cinematography system. We therefore added the Vector3 object which represents a 3-dimensional vector and provides many of the functions needed, including vector and scalar incrementing, calculating the magnitude, rotating one vector around another, calculating the dot product and cross product, and calculating the mid-point between two point-vectors. We also provide a common data structure with the LinkedList object which is used by `EstablishCam` to record placements at which it has already filmed an establishing shot to ensure that the same area is not established twice.

5.6 Problems Encountered

In this section we discuss considerable problems we encountered in the implementation of this project including a lack of documentation for the Quake II engine, finding a suitable place to "plug in" to the engine, separating the position of the player's view in the game world from that of the avatar, and detecting rooms.

When the Quake II engine was released under the Gnu Public License (see Section 5.1) it came with no documentation on the operation of the various components of the engine or the relationship between them. A small amount of relevant documentation could be found on the World Wide Web, mostly on forums, but nothing substantial. Therefore, we had to study multiple lines of code to discover the structure of the engine and game DLL. However, here we found another problem: Comments in the code were few and far between and generally insufficient. Therefore, the only things to go by, so to speak, were the names of functions, variables and so on and even these were not generally consistent. This process which involved creating suitable documentation for use in this project

consumed a considerable fraction of the project time and it was during this phase of the project that we clarified the (inter-)operations of the server, game DLL and client.

While creating this documentation we were also looking for a place to plug in to the game engine, whether that would have been in the game DLL or the client. As it turns out we had to plug in to both but in discovering that we had to contest with obscurely named structs, struct variables and so on to try to discover the ones that were relevant. In the end we found the variables that represent the player's view, those that represent the avatar and so on and designed the GCI and Avatar such that it abstracted the cinematography code from these. Another considerable problem we had to solve, as already mentioned, was to separate the player's view from the avatar's position for third-person `CameraBots` (i.e. most of them). As discussed in Section 5.2 the solution was to create another entity to represent the avatar and to make the avatar invisible. The avatar would now be invisible to the enemies in the game and they would attack, instead, the new entity.

Writing the room detection function was another of the most time consuming tasks in developing the cinematography system due to the trial of a number of different approaches. This is only a reflection of its importance, as without it establishing shots could not be made and the principle of introductions from cinematography could not be applied. One of our early ideas was to perform some pre-processing during the starting up of the game which would involve tracing paths throughout the entire game level and detecting where corridors ended and rooms began. At this stage we were considering detecting not only the avatar's movements from room to room but those of all of the characters in the game. Once we decided room detection was only needed for the avatar, we decided that the calculations should be performed as the avatar moves around the game or "on the fly". It was also considered to actually place triggers in the game levels using a level design tool but this approach was proving too time consuming and was abandoned. In the selected approach, the difficulty in designing the algorithm was choosing which directions to perform traces in, what distances to use and how many to perform.

In this chapter we have presented the aspects of our work specific to its implementation and the problems encountered therein. In the following chapter we conclude on the project as a whole.

Chapter 6

Conclusions

In this chapter we summarise our main points, compare our achievements with our initial aims, provide a general assessment of this work, discuss possible future work, and conclude on a number of points.

6.1 Summary

We began this thesis by making the case that the camera work in 3D computer games could be more sophisticated than at present generally and that cinematography should be looked to as part of the solution. We put the problem in the context of computer graphics, computer games, and cinematography and put forth our specific aims. We discussed a selection of aspects of cinematography that we deem to be suitable for putting into use in 3D computer games including shots, lens height, subject angle, composition, continuity and cutting, and methods for putting all of these together in the shooting of scenes. Next we discussed previous work pertaining to camera control in a 3D virtual environment which we found useful in the development of our own work. This included that of Christianson et al. (1996) and Ting-Chieh et al. (2004) with regard to the application of cinematic shots to a 3D virtual environment (although the work of Christianson et al. is performed off-line and is therefore unsuitable), and that of Amerson and Kime (2001) in terms of the use of a cinematographer module. Next we presented our cinematography system: the `Cinematographer`, `CameraBots` and their various update cycles. We discussed the `Cinematographer`'s editing role, the necessity for freeze frame, the placement algorithms for each `CameraBot`, and examined the algorithm for room detection which is needed to film establishing shots. The results of our testing phase and

of the questionnaire were presented next where it was seen that on our principle measure of success, i.e. the number of enemies the player fragged per unit time, our system performed very well whereas on the secondary measure, i.e. the number useful items the player picked up per unit time, not as much. Then we discussed the specifics of the implementation of our virtual cinematography system including the technology of game engines generally, the Quake II game engine in particular, how it updates the game world and how our system interfaces with it via the GCI and the Avatar object. Lastly, we considered noteworthy problems that were encountered in the development phase of this project.

6.2 Achievements

To assess our achievements we again state our original aims to develop a virtual cinematography system for FPS games that should be capable of:

- Providing more relevant views of the game content that are not limited to the player's point of view
- Varying the level of subjectivity throughout the game
- Adding dramatic emphasis where necessary

If successful, the system should do the above in such a way that:

- It is consistent with cinematography practice
- It has minimal impact on the performance of the player
- It enhances the game-play experience for the player

In addition to this some practical objectives were to:

- Implement the system in such a way that it can be easily incorporated into any relevant game engine
- Implement the system such that it can be easily extended with new features.

Although evaluating some of these aims can be quite subjective, here we will attempt to do so. FPS games provide only a single moving viewpoint in the game. It is rigidly

connected to user input and so is not responsive to developments in the game such as entry into a new setting or the appearance of new characters. We provided a number of active cameras in Quake II that do respond to the change in location of the avatar and the characters in a given setting. For general movement around the game there is `NaviCam` which grants the player a slightly more objective view than the standard first-person view in that he can see the avatar in relation to the scene while the view still moves with the avatar. `EstablishCam` gives the player a general objective view of a new setting allowing him to find the locations of the avatar and other characters in the setting relative to the setting and to each other. `CharacterCam` moves in for more subjective views of characters in the new setting.

In our analysis of important events in Quake II, which would allow us to create `CameraBots` to cater for them and therefore provide more relevant views, we chose the most significant one of the avatar firing at an enemy, but specifically when well aimed. Here `MissileCam` adds dramatic emphasis to such an attack by filming both missile and target at an angle as close as possible to right angles to the direction of motion of the missile with the game in slow motion. Now the player can watch either the success or failure of her well aimed shot. To ensure that our system was consistent with cinematography practise, we considered a number of cinematography principles and guidelines and chose to apply a selection of shot types and the principle of introductions because of its universal applicability. Our fifth aim was obviously a point of concern to us, i.e. that our camera work might impair the player's ability to succeed in the game, but from our results it appears that this is not the case. Participants in the test performed better with our virtual cinematography system operational than while playing the standard version under our first measurement but slightly worse under our second. They also said in our questionnaire (on average) that playing with our virtual cinematography system was a more interesting experience (see Table 4.2) which is positive with respect to our sixth aim. With regard to our seventh aim, our cinematography system is completely independent of the underlying game system; the two communicate through a well defined interface in the form of the GCI and the Avatar object; and it is only this interface that would have to be modified for the system to be applied to another game engine. Even at

that, not every member function has to be redefined. Finally, our eighth aim was addressed by the object-oriented nature of our design. This means that extending the system for application to different games and different game genres is relatively easy. All one needs to do is create new `CameraBot` types and modify the `Cinematographer`'s decision loop to include the new `CameraBot` in editing decisions and repositioning commands

6.3 Project Assessment

This project fulfilled its initial aims by virtue of the development of a virtual cinematography system building on a number of previous works and incorporating suitable cinematographic principles and guidelines. The system is both portable and extensible leaving it open to considerable future work which we consider next.

6.4 Future Work

During the development of this project there were a number of features that we had to abandon due to time constraints and some that were simply outside the scope of the project. These features could be carried out in future work. One area is that of game events. Our implementation responds to the game events of:

- Avatar entering a room
- Characters (enemies) existing in the room
- Avatar facing an enemy
- Avatar firing a projectile weapon at an enemy

It could easily be extended such that more events are responded to. A `CameraBot` could cater for the avatar picking up an item for example or for the avatar performing a particularly dangerous leap. A common feature that could be added to all `CameraBots` is a handheld camera effect. One use of this would be to make the active `CameraBot` become shaky when the avatar's health drops below a certain level. `CharacterCam` might be extended to film close-ups, e.g. matched close-ups of the avatar and an opponent engaging in combat (see Section 2.2).

We had originally planned to take the view outside the room occupied by the avatar so that `CameraBots` could film other rooms and their occupants subject to certain criteria, including, perhaps, the number of enemies in an adjacent room. This would require expansion of the room detection method and perhaps some pre-processing as mentioned in Section 5.6. Other possible extensions include adding panning and tracking to the `EstablishCam`, `CharacterCam` and `MissileCam` such that sometimes still shots, other times panning shots, and on other occasions tracking shots are filmed. This would add to the visual variety of the experience.

Three other significant additions to the system would be adaptation for action replays, spectator views and level walkthroughs. Action replays could be applied to Quake II and other FPS games with `CameraBots` such as `MissileCam`: the player would get to see a replay of a particularly good attack, for example. Spectator views would be suitable for multiplayer games and tournaments where individuals view spectator versions of a given game. Here the `Cinematographer` would have much more freedom in filming different settings by using cross-cutting for example (Section 2.6) and applying a greater set of cinematographic principles. Walkthroughs similar to that of Nieuwenhuisen & Overmars (2003) could be performed for game levels, as mentioned, as an introductory device as an alternative to the use of `EstablishCam` and `CharacterCam`.

Overall, the entire virtual cinematography system could be used with different game engines, which means that it could be applied to a number of different game genres since different sets of games will have been developed with different engines. This implies further research into cinematography to select appropriate principles and guidelines for a given game genre. Lastly, we did not consider the principles of lighting granted by cinematography in our work and obviously lighting has implications in shooting any type of action. The addition of these principles to a virtual cinematography system has the potential to enhance the dramatic impact of computer games by another factor.

6.5 Conclusions

We conclude on whether further research should be conducted in the area of virtual cinematography and evaluate its commercial applicability. We believe that in the way that a different set of camera operations is applied to each different genre of motion picture so too should there be a different set of camera operations, principles and guidelines for the various genres of computer games that exist. The implication of this is that many extensions to our virtual cinematography system would be justified. We also consider the idea of a cinematography module to make a great deal of sense as part of a general-purpose game engine or perhaps of a middleware solution for game developers. Our reasoning is that it should not be up to a game developer to gain a considerable knowledge of cinematography in order to design a camera system and it seems unrealistic to attempt to develop a new virtual cinematography system for each individual game especially when numerous `CameraBot` types and editing decisions would most likely be suitable to numerous games and would therefore be more suitable in a general-purpose system.

References

1. **Amerson, D. & Kime, S. (2001).** 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.

2. **Blinn, J. (1988).** Where am I? What am I looking at? IEEE Computer Graphics and Applications, 8(4): 76-81.

3. **Brown, B. (2002).** Cinematography: Image Making for Cinematographers, Directors and Videographers. Oxford: Focal.

4. **Charles, F., Lugrin, J. L., Cavazza, M. and Mead, S. J. (2002).** Real-Time Camera Control for Interactive Storytelling, 3rd International Conference on Intelligent Games and Simulation, pp. 73-76.

5. **Christian, D. B., Anderson, S. E., He, L., Salesin, D. H., Weld, D. S. & Cohen, M. F. (1996).** Declarative Camera Control for Automatic Cinematography. In Proceedings of the Thirteenth National Conference on Artificial Intelligence, pp. 148-155.

6. **DeMaria, R. & Wilson, J. L. (2002).** High Score! The Illustrated History of Electronic Games. McGaw-Hill/Osborne.

7. **Dijkstra, E. W. (1959).** A note on two problems in connection with graphs. Numerische Mathematik, pp. 260-271.

8. **Eidos Interactive. (1998).** Tomb Raider III, computer game for IBM compatible PCs, released by Eidos Interactive, San Francisco, CA.

9. **Eidos Interactive. (2002).** Hitman 2, computer game for IBM compatible PCs, released by Eidos Interactive, San Francisco, CA.

10. **Fairclough, C., Fagan, M., Mac Namee, B. & Cunningham, P. (2001).** Research Directions for AI in Computer Games. Proceedings of the Twelfth Irish Conference on Artificial Intelligence and Cognitive Science, pp. 333 – 344.

11. **Foley, J. D., van Dam, A., Feiner, S. K. & Hughes, J. F. (1990).** Computer Graphics, Principles and Practice, Second Edition, Addison-Wesley Publishing Company.

12. **Friedman, D. and Feldman, Y. (2002).** Knowledge-based formalization of cinematic expression and its application to animation, in Proc. EUROGRAPHICS 2002, pp. 163–168.

13. **Funge, J. (1999).** Cognitive modeling for computer games. AAAI Spring Symposium on Artificial Intelligence and Computer Games, Stanford University.

14. **Gnu.** http://www.gnu.org/

15. **Graham, R., McCabe, H. & Sheridan, S. (2003).** Pathfinding in Computer Games. ITB Journal Issue Number 8, December 2003.

16. **Graham, R., McCabe, H. & Sheridan, S. (2004)**. Neural Networks for Real-time Pathfinding in Computer Games. ITB Journal Issue Number 9, December 2004.

17. **Halper, N. & Olivier, P. (2000).** CAMPLAN: A Camera Planning Agent. In Smart Graphics, Papers from the 2000 AAAI Spring Symposium, pp. 92-100.

18. **Halper, N., Helbing, R. & Strothotte, T. (2001).** A Camera Engine for Computer Games: Managing the Trade-Off Between Constraint Satisfaction and Frame Coherence. In Proceedings of Eurographics, pp. 174–183.

19. **He, L., Cohen, M. F. & Salesin, D. H. (1996).** The Virtual Cinematographer: A Paradigm for Real-time Camera Control and Directing. In Proceedings of SIGGRAPH 1996.

20. **id Software. (1997).** Quake II computer game for IBM compatible PCs, released by id Software, Mesquite, TX.

21. **Laird, J. & van Lent, M. (2001).** Human Level AI's killer application: Interactive computer games. AI Magazine, Volume 22, Issue 2.

22. **Laird, J. E. (2000).** It Knows What You're Going To Do: Adding anticipation to a QuakeBot. AAAI 2000 Spring Symposium on Artificial Intelligence and Interactive Entertainment. AAAI Technical Report SS00–02. Menlo Park, CA: AAAI Press.

23. **Laird, J.E. & Duchi, J.C. (2000).** Creating Human-Like Synthetic Characters with Multiple Skill Levels: A Case Study Using the Soar Quakebot. AAAI tech. report, SS-00-03, AAAI Press, Menlo Park, Calif.,

24. **H. Levesque, F. Pirri, and R. Reiter (1998).** Foundations for the situation calculus. Electronic Transactions on Artificial Intelligence, 2(3-4):159-178.

25. **Luger, G. & Stubblefield B. (1998).** Artificial Intelligence: Structures and Strategies for Complex Problem Solving, third edition

26. **Mascelli, J. V. (1965).** The Five C's of Cinematography. Los Angeles: Silman-James Press.

27. **Nieuwenhuisen, D. & Overmars, M. H. (2003).** Motion Planning for Camera Movements in Virtual Environments.

28. **Reynolds, C. (1999).** Steering Behaviors For Autonomous Characters. Game Developers Conference 1999.

29. **Sánchez-Crespo Dalmau, D. (2004).** Core Techniques and Algorithms in Game Programming. Indianapolis, IN : New Riders Publishing.

30. **Švestka, P. (1997).** Robot motion planning using probabilistic roadmaps, PhD thesis, Utrecht Univ.

31. **Ting-Chieh, L., Zen-Chung, S. & Yu-Ting, T. (2004).** Cinematic Camera Control in 3D Computer Games. In Proceedings WSCG 2004.

32. **Tomlinson, B., Blumberg, B. & Nain, D. (2000).** Expressive Autonomous Cinematography for Interactive Virtual Environments. In Proceedings of the Fourth International Conference on Autonomous Agents, pp. 317-324.

33. **Ubisoft (2003).** Prince of Persia: The Sands of Time, computer game for IBM compatible PCs, released by Ubisoft, Austin, TX.