Automated Cinematography for Games

a thesis submitted to Middlesex University
in partial fulfilment of the requirements
for the degree of Master of Philosophy

Laurent Cozic
Lansdown Centre for Electronic Arts, School of Arts
Middlesex University

London, June 2007

Acknowledgements

I would like to thanks my supervisors Stephen Boyd Davis, Magnus Moar and Gordon Davies for their support and advice through this MPhil.

Contents

I. Introduction _______ *5*
1. Methodology _______ 6
2. Rationale _______ 7

II. Background _______ *15*
1. Existing approaches _______ 15
2. Camera Systems in Games _______ 18
3. Current Issues _______ 20

III. Camera System Description _______ *22*
1. Proposed Camera System _______ 22
2. Constraint Generator _______ 25
3. Constraint Solver _______ 28

IV. Camera System Development _______ *29*
1. Software Used _______ 29
2. Story Engine _______ 31
3. Camera System _______ 33

V. Conclusion _______ *48*

VI. Bibliography _______ *50*
1. Literature _______ 50
2. Websites _______ 51

VII. Games _______ *52*

VIII. Source Code _______ *53*
1. Story Engine _______ 53
2. Camera System _______ 53
3. Knowledge Base Sample _______ 64

IX. Glossary of Cinematographic Terms _______ *68*

X. The Intruder – Expressive Cinematography in Videogames _______ *70*

Abstract

This thesis deals with the issue of automated cinematography for games. In 3D videogames, the system must continuously provide the player with a view of the virtual world and its characters. The difficulty is that contrary to the cinema the actors are unpredictable. In particular the player continuously modifies the virtual environment by moving objects or by interacting with the other non-playable characters. The latter, because of their more and more sophisticated artificial intelligence, can have behaviours that were not predicted by the developers themselves (such as the complex behaviours that emerge from the combination of basic behaviours).

Some games have solved the problem by predefining the possible positions of the camera during the game development while some others give control of the camera system to the player, so that he can find by himself the best possible view. I aim however at finding an intermediate solution, where the camera system would automatically generate both engaging and usable views. The camera system should be able to adapt to every situation of the virtual world without user intervention, and should allow the player to interact with his surrounding in the most efficient way. Such a camera system could be of interest for the game industry. Currently, in many games, the camera movements, positions, etc. are set using scripts manually written by the developers. Having a fully automated system could potentially save hours of work. This system could also be used for the 3D virtual worlds or "3D chats" on the Internet. For example, the avatars – the characters played by the users – could be "filmed" in a different way depending on the mood of the users. I aim to develop techniques which can be generalised to these and other areas of application.

Existing approaches to automated cinematography will be reviewed – focusing on the constraint-based and idiom-based ones – in order to highlight the strength and limitations of each one. A solution to the problems found will be proposed in the form of a camera system implemented using Adobe Director. It will be based on "rules" derived from existing cinematographic knowledge. One of my aims will also be to show that using generic rules can give results close to the idiom-based approaches with the convenience of being able to adapt to any type of scene.

I. Introduction

This thesis comprises the following main parts:

I. Introduction
II. Background
III. Camera System Description
IV. Camera System Development
V. Conclusion

In this thesis, I will focus on finding solutions to reconcile the dramatic potential of cinematography with the demand of an unpredictable interactive experience. Most camera systems attempt either to create cinematographic shots or to create playable shots but never both at the same time. It appears that to deal with one of these two aspects the other has to be somewhat neglected. It will be demonstrated through this thesis that it is possible to deal with both aspects simultaneously. The idea will also be presented that it is possible to create a game that seems "fairer" to the player by using cinematographic shots.

In the first part of this thesis, I explain the overall aims, namely reconciling the dramatic potential of cinematography with the demands of an unpredictable interactive experience. The film-maker's achievement will be defined, and the aspects of it that will be used for that camera system will be highlighted. The control of the camera in videogames will then be discussed, and show why it should be the task of the game engine and not the player. On one hand, my argument will be based on the fact that the less controls there are to handle, the more playable the game is. On the other hand, the game can be improved if the game designer can have a rhetorical control over the generated shots.

In the following part, the existing approaches to automated cinematography will be reviewed and the strengths and limitations of each one will be outlined. In particular, the focus will be on the two most important types of techniques: the constraint-based and the idiom-based. I will also describe the different existing camera systems in third-person games and show clearly the different degrees of freedom they give to the player. To conclude this first part, the main issues that are shared by most of these systems will be outlined: the fact that they often lack the flexibility to adapt to the vast virtual worlds of modern games and the difficulties they have to conciliate playability and cinematographic viewpoints.

The solutions that have been found to these problems will then be described. Especially I propose that the camera system should be more connected to the story engine, in such a way that it "knows" what should be in view and what could be ignored. The strength of such a system and the way it can be implemented will be outlined. Some parts of it will be described in more details, especially the type of cinematographic rules that will be used and how the constraint generator and the constraint solver will work.

The next part of this thesis will describe how the camera system has been implemented. It has been developed using 3D Studio Max and Adobe Director. Each of them will be briefly reviewed in introduction. The structure of the story engine will also be described as it is strongly connected to the camera system. The different parts of the camera system will be detailed: the structure of the knowledge base and the different steps followed by the camera systems when creating a new shot. Finally, the system behaviour will be analyzed using a specifically designed test bed. An account of the strengths and weaknesses of it in each situation will be given.

1. Methodology

In order to gather information during this thesis, I have made a review of existing camera systems by studying various games. I considered mainly third person view adventure and action games. Examples of what I would call an adventure game include *Resident Evil* (1996) or the *Monkey Island* series (1990). Typically, these games are mainly based on the resolution of puzzles and on an elaborated plot. What I will call action games by contrast appeal to the player's dexterity and reflexes. The main character has many more possible movements. The plot is poor or nonexistent as it is not the main concern of the game. Typical examples include *Super Mario Sunshine* (2002) or *Tomb Raider* (1996). Some games also belong to both genres such as *The Legend of Zelda: The Wind Waker* (2002) which alternates between action and adventure sequences.

I have also studied the various papers that have been written on automated cinematography. They will be fully(!) reviewed in *II. Existing approaches* below. As part of my investigation, I have also contacted a few other researchers that have been or that are still involved in automated cinematography. Alexander Hornung and Doron Friedman have given me additional details on their research and have pointed me out to some useful papers that I had missed. To design the camera system, I have also relied on a number of texts on film-making. This MPhil being a continuation of work originally intended for an MA, part of it will also be based on previously written material that will be referred back to. It includes a review where I had listed a number of editing techniques and the way they can be applied to video games. These ideas have been used to develop the camera system rules and the logic behind them.

I have also read a few texts on basic artificial intelligence techniques. Among others, I had a close look at the Artificial Intelligence Markup Language (AIML), which I ended up adapting to make the camera system knowledge base (See *IV. AIML* for a description of the language).

In order to experiment with some of the concepts used in my camera system, I have also made two test programs. One of them to experiment with visual constraint solving: the program outputs camera coordinates (its rotation and position) depending on the requested visual constraints (The program is described in *IV. Camera System Development: Constraint solver implementation*). To experiment with decision-making algorithms, I have also made a simple expert system that advices on which jacket should be worn depending on various factors.

2. Rationale

Virtual cameras can do a number of things that physical cameras cannot. Thus, one might wonder why we should try to reproduce techniques that are limited by physical world constraints. After all, there is technically no limit to what can be done with a virtual camera – it can move at any speed, go through the walls or be positioned anywhere and with any angle.

Despite this, it is still worthwhile to try implementing existing cinematographic techniques in game camera systems. The reason is that, over more than a hundred years, film-makers have developed techniques to present and articulate drama. We can therefore assume that it is worthwhile to try to bring together this knowledge and the demand of interactive games. Additionally the goal of most modern games is to appear more and more convincing. To achieve that goal the experience of film-makers is also valuable.

The system I wish to develop will be influenced mainly by mainstream fiction film tradition. These films usually try to appear naturalistic. In other words, the viewer can experience the whole movie without having his attention drawn to the fact that it is only a representation. In these films it is considered as undesirable to break the suspension of disbelief.

To achieve that apparent naturalness, filmmakers use various means. At a simple level, sticking to rules of cinematography, such as those described by Roy Thompson (1993), will ensure that the story will be followed smoothly by the viewer. For example, one of the rules is that the camera should not cross the "line of action" during an edit (Please refer to the glossary in section IX for a definition of each cinematographic term used in this thesis) in order for the viewer not to notice the cut between two shots.

In general, strictly following these rules will allow a film editor to achieve good continuity. Thus Karel Reisz argues that "the main purpose of assembling a rough cut is to work out a continuity which will be understandable and smooth. […] Making a smooth cut means joining two shots in such a way that the transition does not create a noticeable jerk and the spectator's illusion of seeing a continuous piece of action is not interrupted."

The apparent naturalness of a movie also depends on cultural factors. A technique that has been used for one-hundred years is less likely to be noticed that a brand new visual effect. For example, Boyd Davis (2002) mentions that the close-up shot used to be shocking for the audience when it was first used. Today it is such a common technique that the viewer will not notice it as such.

However, it is important to notice that the choice of a close-up over for example a long shot will have a different psychological effect on the viewer. It will for example give a clue on the emotional state of the character and therefore psychologically affect us. Thus a shot is not only used to inform but also to influence the viewer, which is why it has been called by Harrington (1973) a form of rhetoric. Even if the film seems natural to the viewer and the techniques used *per se* are not noticed, it will still have a psychological impact on him.

- *Definition of a playable / usable view*

In this thesis, I will consider that a playable view is one that is *fair* to the player. The idea is one can play the game without risk of losing the game through an ill-positioned camera – for example, one that hinders a hazard he should logically have seen. Thus a playable view allows making informed decisions in that it

limits the number of random actions needed to complete the game (such as having to jump in the direction of a platform that is not yet visible on screen). Eventually it should make the game more engaging as it allows the player finishing it out of his own ability and not out of luck.

Strictly speaking, this fairness is only meant to assess the playability of a shot or a series of shots. It is possible to get to get a good estimate of it by playing through the game and considering the two following criteria:

1) Can the player lose the game because the camera moved by itself at the wrong moment? This is sometime a problem in platform games. For example, the player's character might fall if the camera rotates around him at the very moment he is jumping from one platform to another.

2) Can the player lose the game because an enemy that he/she should have seen was occluded by another object? This frequently occurs in the Resident Evil games, where the zombies are often occluded by the corners of the corridors. Obviously, in some games, the enemies might hide themselves on purpose. For example, in most recent "doom like" games, the enemies behave like normal soldiers and hide themselves in order to avoid being shot. However we only deal with more obvious failures of the camera system such as not showing an enemy that is right in front of the character. Basically, we can say that there is a problem if the player cannot see something that his/her character can clearly see.

Comparing these two criteria to the total number of times the player's character has been hit should give a relatively good idea of the fairness of a camera system

Finally, it is worth mentioning that in some instances, it might be good to have "unfair" shots on purpose in order for example to increase the difficulty of a game. It is undeniable that the mood and tension of the early Resident Evil games own a lot to the fact that nearby enemies are often hindered because of a "wrongly" positioned camera. It might seem unfair but it adds a lot to the feeling of being in danger. However, in this thesis I will only deal with fair shots as defined here, noting that if a perfectly fair camera system was to be done it would also be possible to set it so that it creates, if needed, "unfair" shots.

- *Scope of the cinematography aspect of the camera system*

Cinematography refers to a number of choices such lighting, lens choice, filtering, etc. In other words, it refers to all the aspects that might affect the appearance of an image. However, the camera system presented in this thesis deals only with two aspects of cinematography. It deals with *shot composition* – for example, which characters should be on screen and under which angle they should be filmed. It is a task similar to the work of a cameraman. Secondly the camera deals with *shot selection* – as a film editor, the system decides which shots should be selected among the many possible ones. For the sake of simplification, when I use the terms "cinematographic shot" in this thesis, I will mean shots that deal with these two aspects of cinematography.

• *Composition*

Fig. I.2. Stanley Kubrick: *The Shining*, 1980. Introduction shot showing the main hall.

According to Gessner, a "shot is composed of frames […] and should aim to express a single cinematic emotion or idea." Film-makers compose each shot in such a way as to allow the viewer to make sense of the film space. Scenes are sometime introduced by an establishing shot – a shot that shows everything, characters and scenery included, in the current scene. The way each shot is composed is driven by the narrative. The camera angle, its distance to the characters will depend on what the director wants to say about them. For example, the shot in Figure I.2 from *The Shining* (Kubrick 1980) introduces the viewer with the main hall, where a good part of the film will take place – thus giving the viewer the necessary spatial information to make sense of the subsequent scenes. By playing on the relation between the size of the big room and the comparatively small size on screen of the character, the shot may also aim at highlighting the central character isolation. In other words, the filmmaker knows where the things that matter in the diegetic world are, and therefore knows what the best way to show them is. In a similar way, a video game camera system needs to have access and need to make good use of the available narrative information to make a good shot composition. One of my aims in this dissertation will be to show how to achieve this task.

The composition of a shot can be described using seven variables listed in the table below.

Variable	**Static**	**Dynamic**
Location of camera	Location	Track, dolly, pan, etc
Target	What is centre-screen	Changes of target
Lens size	Wide angle or long, etc.	Zoom
Focus	What is in focus	Pulling focus
Depth of field	How much is in focus	Altering depth of field
Optical	Filters, film stock etc	Altering say colour to b/w
Tilt	Tilted shot	Altering camera tilt

Table 1. Variables of cinematography.

- ***Shot Selection***

A film-maker constantly has to choose shots among many possible ones. All of them might be interesting in terms of composition or in terms of the information they give to the viewer. Which shot will eventually be selected depends on the following factors:

In the first place, a film-maker will choose a particular shot depending on where he wants the narrative to go. In other words, through the selected shots he will show the viewer what he needs to see or know to be able to understand easily the movie. Carroll notes that "the movie spectator is always looking where he or she should be looking, always attending to the right details and thereby comprehending, nearly effortlessly, the ongoing action precisely in the way it is meant to be understood."

A shot usually needs to be shown in context to have the intended effect; therefore, the previous shot(s) will have an influence on the selection of the current shot. For example, after an important event occurs, the film-maker might choose to show a reaction shot of a one of the witnesses to heighten the effect of the scene. The reaction shot in itself would be irrelevant – it has only been selected because of the context.

Choosing a proper sequence of shots also helps the viewer to build a clear mental picture of the diegetic space. This is true both in cinema and videogames. For example in Fig. I.3 below, each shot has been chosen in such a way as to help the player to make sense of the police station room more easily. Among other things, this is achieved by having elements shared between shots (i.e. elements such as the door from shot I to shot II, then the ventilator from shot II to shot III help to link mentally the shots) and also by respecting rules of cinematography. For example, in this sequence the camera is kept on the same side of the "line of action" in order not to break the continuity of shots.

Fig. I.3 (Shot I, II, III). Capcom: *Resident Evil 2*, 1997. The camera stays on the side of the "line of action" in order not to confuse the player.

A motivation for showing a new shot can also be to avoid monotony and to create visual diversity. Technically a conversation could often be filmed with a single long shot of the characters involved. It would provide the viewer with the necessary information about the topic discussed. However most film-maker will avoid this and make the scene more dynamic by alternating with long shots, close-up, etc. or by cutting to other events happening in the surrounding. However, although the first motivation for this visual diversity might be to make less monotonous scene, each shot will rarely be selected completely randomly. Whether a shot shows the reaction of a character, or the scene surrounding in order to set the mood, the selection is still largely driven by the narrative.

When dealing with shot selection in this thesis, I use the concept of the "optimal view". Boyd Davis (2002) describes it as the view that is the most expressive in terms of information and affect. For example, a close-up might be expressive in terms of information, as it allows the viewer seeing necessary details, such as what is the character's frame of mind. In terms of affect, the shot might also startle, alarm or engage emotionally the viewer. By selecting only optimal views, the film-maker will ensure that his story will be well understood as the viewer will always have all the necessary information, and all the significant events will always be in view.

However, Boyd Davis also demonstrates that there are limits to the optimal view: a film-maker might deny it on purpose in order for example to hold the suspense. In this series of shot from *The Shining* (Fig. I.4), the boy moves towards a room that has been mentioned several times before during the movie, and which is now mysteriously opened. At the precise moment when the boy is about to enter it, the film suddenly cuts to a new, apparently unrelated shot. By denying the optimal view, the film-maker holds the suspense – and leaves room for interpretation as we will never know what actually happened there – thus creating a more engaging story.

Fig. I.4 (Shot I, II, III, IV). Stanley Kubrick: *The Shining*, 1980. Although the viewer wants to know why the room 237 is suddenly opened and what is going to happen next (Shot I, II and III), the film suddenly cuts to something unrelated in shot IV.

- ***Camera control in video games***

All camera systems in games include some sort of automation, in that none would fully allow the player to control the six degrees of freedom of the camera (Its three X, Y and Z coordinates and its rotation around the X, Y and Z axes). The camera movements will usually be at least restricted to a number of sensible positions or orientations. For example, in *Super Mario 64* (Nintendo 1996), the camera automatically points at the player's avatar and can only be rotated around it. The level of control usually depends directly on the level of precision required by the game play. In *Super Mario 64*, the player has four buttons to control the camera, to allow precise positioning. Pressing the Up and Down buttons will move the camera towards and away from the character, while pressing the Left and Right buttons will rotate it by 45° increments. The game can indeed require great precision to avoid obstacles or to jump from one platform to another. In comparison, only one button is used to control the camera in *The Legend of Zelda: Ocarina of Time* (1998): it resets its position so that it points in the same direction as the

character. Indeed, it is not a fast-paced game and therefore precision is less of an issue. Additionally only a few movements are irreversible so that a wrong camera position will not have a crucial impact on the game progress. It may temporarily confuse the player but eventually he will not lose the game because of it.

For the camera system I wish to achieve, player input will be kept to a minimum. Ideally the camera will be able to track the player with no input at all. There are two reasons for which I would like to limit the control of the camera: the first is playability, the second is because of the loss of rhetorical control from the designer:

- *Playability*

Games can quickly become very complex due to the number of possible interactions they offer. The *DualShock* controller (Sony 1997) has eight main buttons (The Triangle, Square, Circle and Cross buttons, as well as the four shoulder buttons), all of which are sometime used to control the playable character. If the player also has to control the camera on top of it, it can lead to too much complexity – to the point where one has to focus more on the camera control than on the game itself. This issue appears for example in *Super Mario Sunshine* (2002). In this game, both the button to jump and the joystick to orientate the camera are controlled by the same finger (the right thumb). However to be able to go through some passages, it would be necessary to both jump and control the camera at the exact same time, which is physically impossible (Fig. I.5). It is still possible to go through these passages but only by knowing them by heart and by jumping "blindly", as the camera will not show automatically what the player needs to see. These instances being quite rare in the game, we can assume that it is not done by design (the control of the camera was not meant to be a part of the challenge) but a problem arising from the fact that the camera is not fully automated.

Fig. I.5. (a) Nintendo: *GameCube* controller, 2001 and **(b)** *Super Mario Sunshine*, 2002. The "A" button is used to jump and the yellow joystick to control the camera in *Super Mario Sunshine*. Both are controlled with the right thumb. It leads to odd situations when the game requires the player to jump from platform to platform while simultaneously controlling the camera in order to make sense of the surrounding.

Therefore, one of the motivations to fully automate the camera is to simplify the player's task. Having less buttons to consider, the player can then concentrate on the gameplay itself and "forget" about the controller he is holding. Allard (2005) stated at the 2005 Game Developers' Conference that "controllers are incredibly important, and the important thing about control, is that it becomes invisible." The forthcoming Nintendo controller (Fig. I.6) also follows that direction. It only has four main buttons and can simply be held with one hand.

Fig. I.6. The controller of the forthcoming Nintendo Revolution in 2006. The current tendency in the game industry seems to be to simplify to the maximum the controller. It comes from the assumption that the less the player has to *think* about the controller he is holding, the more he can focus on the gameplay. To that respect, removing the need for camera control will allow simplifying the game control even more.

- *Rhetorical control from the designer*

The game designer knows what the player needs to see to play the game; he is therefore able to set the behaviour of the camera in such a way as to show the most playable view at any given time. This is for example subtly done in a few levels of *Super Mario 64* (1996). In the first level (Fig. I.7), the camera follows the character from behind but also automatically rotates around him to show the main path. This feature allows the player to complete the level without having to control the camera at all. The designer knowledge of the level has been applied to the camera in order to lighten the player's task.

Fig. I.7. Nintendo: *Super Mario 64*, 1996. The camera automatically shows the recommended path without any player input in the first level of Super Mario 64.

If a game designer has a rhetorical control over the camera, he is also able to create views that are more engaging. This is obvious in games such as *Resident Evil* (1996) where the predefined views serve the narrative. Views are often chosen to heighten the game tension (by purposely giving bizarre camera angles for example) or to highlight certain important objects. Establishing shots are also used when the player enters a new place, such as the very first view in *Resident Evil*, which shows the main stairs and most of the important places the player has to explore at the beginning.

- ***Virtual Camera***

Most if not all of the aspect of a real camera can be emulated in today's game engines. However, I only aim at dealing with the camera *location*, *target*, *tilt* and *lens* variables. In a game engine, the location of the camera is determined by its X, Y and Z coordinates. The target of the camera will usually be determined by its rotation around its X and Y axis, and the tilt by its rotation around its Z axis. In photography, the field of view depends on the lens size; however, in this thesis I will deal directly with the more explicit field of view. The table below shows the variables of cinematography (cf. Table 2) I am dealing with and how they relate to a virtual camera. All these variables can be used statically or dynamically. In the coordinate convention I am using, Y is up and Z across the camera plan.

Variable	**Determined by**
Location	X, Y, Z *position*
Target	X, Y *rotation*
Tilt	Z rotation
Lens size	*Field of view* (FOV) value in degrees[1]

Table 2. Real world camera variables (left) and their equivalents for a virtual camera (right).

[1] The aspect of a shot will also depend on the pixel aspect ratio. In this thesis, it is assumed that the screen pixels are square.

II. <u>Background</u>

1. Existing approaches

Different approaches have been used to automate the camera movements in virtual environments. Drucker and Zeltzer (1995) argue that in a virtual environment the user should not have to control the six degrees of freedom of the camera because it would prevent him from concentrating on his task. In order to avoid that they have encapsulated all the camera tasks into "camera modules", which comprise all the information required to make a shot. The system expresses these modules as visual constraints such as: which size should each character be on screen, and under which angle should they be visible. For example, in order to make a close-up shot, they would simply ask the system to show the head of a character and to ensure it fills 90% of the screen. From this request a constraint solver outputs the parameters for the camera (The X, Y, Z position and rotation). Bares *et al.* (2000) use a similar constraint-based approach in their multi-shot visualization interface*s*. They also propose alternate solutions when the constraints cannot be satisfied whether by decomposing viewing goals into multiple shots or by relaxing weak constraints. They illustrate this technique using a simulation in which several policemen have to catch a thief. The user has to choose which characters he wants to see and in which way. When all the characters are relatively close to each other, a single shot showing all of them is used. However when such a shot is not possible, the system displays a sequence of shots or a composite view of all the shots, depending on the user's preferences. The composite view is made of an overview showing the whole simulation and one or two inset shots. The shot sequence alternates between shots showing one or several of the user selected characters. This technique has the advantage of giving the user an exhaustive view of the virtual world and therefore could be particularly suitable for games such as *Populous* (Bullfrog 1989) or *Sim City* (Maxis 1989) where the user must be aware of everything to be able to play. However it would be less relevant in games more concerned with scenario and character identification, as the "god view" creates feeling a detachment between the player and the virtual world. Additionally as it is the system does not deal with playability. As a result, some of the visual solutions, especially the shot sequences, would not be acceptable for a game as they may prevent the player from seeing his avatar for too long a time.

He *et al.* (1996) use "idioms" to describe the behaviour of the camera for each given type of scene. The rules of cinematography "are codified as a hierarchical finite state machine" which "controls camera placements and shot transitions automatically". The system also uses a library of camera modules, which contains information such as the position of the characters on screen and the shot scale. In its current state, the system can successfully capture specific scenes such as a conversation between two or three actors; however the authors mention that the system can fail due to unexpected occlusions. The other noteworthy limitation is that it can only deal with forecast situations thus making it difficult to adapt it to complex virtual environments. Amerson *et al.* describe a similar model of interactive cinematography called FILM (for Film Idiom Language and Model). They use a tree to encode the knowledge about film idioms with five levels: Generic shot, Shot type, Number of subjects, Emotional effect and some Optional keywords for finest shot selection. The scene is then expressed as a series of constraints, "each constraint [having] a weight, indicating its relative importance to the shot, so that constraints can be appropriately relaxed if all constraints cannot be satisfied simultaneously." The system then uses a pipeline of three software objects to generate the final shots: the Translator, the Director and the Cinematographer. The Translator takes its input from the virtual world and converts it into data intelligible by the Director. The later uses this information to perform a depth search to select the best scene from the tree and send the constraints to the Cinematographer, which output the parameters for the camera.

The idioms are a convenient and straightforward way to encode cinematographic knowledge. In He *et al.* system they can be thought of as functions that take one or more characters as an input, and output a shot solution (for example, the "conversation" script will take two or more characters as an input). I shall review in detail its strength and weaknesses in the *Current Issues* part below.

Halper *et al.* (2001) discuss the importance of having a good balance between constraint satisfaction and frame coherence in order to have smooth camera movements. When a new event changes the current shot, their system ensures that the camera does not jump but smoothly moves to its new location. To deal with camera occlusion they propose a new method based on a projective shadow casting algorithm. From each point of interest in the scene, they cast a light and render the result in a temporary image buffer. They compose a final buffer from all of them in which the lightest parts (in which no object has cast a shadow) are the positions where the camera is less likely to be occluded. With this technique, the camera performs remarkably well when it comes to avoid occlusion and collision. On their website, they show a video demonstration of a teapot flying in a cluttered attic. The camera manages to follow it with little occlusion and with no jerky movement. However, the lack of narrative goal to drive the system sometime prevents it from computing the shot most needed by the player. For example, one of their videos shows a typical medieval adventure game. As in the teapot demo, the camera follows the character without any occlusion. However, when the character enters the house the camera stays outside failing to show the inside of the room. A film-maker would have for example shown a shot of the character entering through the door, followed by a longer shot of the whole room to recontextualise the story. Halper's system however does not as it is only designed to follow the character but does not consider the narrative to compute each shot.

Tomlinson *et al.* (2000) describe a system in which the camera – like all the characters in their virtual environment – is an autonomous agent. A sensor gathers information about the emotions and motivations of the characters. This is combined with the emotional state of the camera and its motivations, which will have an effect on the camera style and motion style. They describe for example the behaviour of happy cameras, which "cut more frequently, spend more time in close-up shots, move with a bouncy, swooping motion, and brightly illuminate the scene". All the possible behaviours compete with each other and an action-selection mechanism will select which one should become active in order to generate the best possible shot. One of the problems of this system is that it is designed to work with their specific autonomous agents, which makes it difficult to adapt it to another virtual environment. However it includes a number of noteworthy features such as the possibility for a character to request a shot and the fact that the camera is not always focused on the user's character. Indeed when an event is important enough, the camera system can insert a shot of it that will not necessarily include the user's character.

Hornung *et al.* (2003) have designed a camera system that improves *Half Life* cut-scenes by choosing cinematographically appropriate shots. Narrative events are sent to the camera module, which chooses an active event for visualization based on the history of the narrative. They applied this technique to the first dialogue sequence of the game. In the original game, the player can only see it in the default first person view; however, their system turns it into a cinematographic scene. It shows for example a long shot of the two scientists talking to each other, then a close-up of one of them and so on. The system automatically creates more appealing cut-scenes for the game; however, it cannot be used in an interactive context

Hawkins (2003) in *Creating an Event-Driven Cinematic Camera* proposes a possible (non-implemented) camera system where he translates the key roles of film industry into entities in the computer (C++ classes). He thus defines a director, a shot, a scene and an editor object. The Director role is to provide the Editor with an up to date list of Shots. Shots are removed from this list when their priority falls under zero or when they are chosen by the Editor object. A problem of this system, as it is described, is that it only uses the position of the actors and the line of action to generate the shots. However this may be

insufficient to generate a good shot: a cinematographer would usually need more information, such as the dramatic character of the scene and the surrounding of the characters.

Friedman *et al.* (2004) made a system called *Mario* that automatically generates animated 3D movies from a screenplay and a floor plan. It uses the multi-layered reasoning system Cake to formalize the cinematographic rules. The system is flexible and allows inputting rules such as "if an actor is speaking, she is displayed in a frontal medium shot"; "if an actor is walking, she is displayed in a long shot". In case of conflicts between the rules, the system uses three types of premises to make a decision: defaults, assumptions and preferences. Defaults are assumed to be true and are the first to be retracted in the case of a contradiction; assumptions are assumed to be true but they do not get retracted automatically by Cake; preferences are similar to default with the difference that they are the last to be retracted. The system seems to perform particularly well for most screenplays. In some cases it automatically generates movies with consistent styles that can be related to particular filmmakers. However the use of a screenplay – a chronological description of past events – makes the system usable only for non-interactive scenes.

Giors (2004) describes the "autolook" function developed for *The Full Spectrum Warrior* (2004) Camera System. Using a ray casting technique, the autolook function determines which part of the view is unobstructed. It then rotates the camera in the direction of the unobstructed part to compose a more playable view. For example in Figure II.1a below, a large part of the view shows a wall against which the soldier is. However, this information should be of little importance for the player. On Figure II.1b, however the camera has rotated to the right, thus "bringing more of the playable area into view".

Fig. II.1 (a-b). THQ: *The Full Spectrum Warrior*, 2004. The camera rotates toward the non-obstructed part of the view.

Christie and Normand (2005) argue that "the description of a cinematic shot can possibly yield different visual solutions". They therefore designed a system that generates a set of possible solutions instead of a unique one. The virtual space is partitioned into "semantic volumes" within which all the possible camera locations will produce "semantically equivalent shots." The system performs as expected, however it is currently too slow to be adapted to a real-time environment. Besides as it produces a *set* of solutions, some additional processing is still required to find the best *unique* solution within that set.

O. Bourne and A. Sattar (2005) developed a constraint-based camera system. They encapsulate the sets of constraints into "camera profiles", which are comparable to He's "idioms". In order to find the best profiles they evolve them using a genetic algorithm. The latter "takes in a set of example animation traces, and evolves specific camera profiles to replicate the movement patterns". The main benefit of this tool is that it can assist the game designer in finding the set of constraints that will the most closely matches the kind of camera style he has in mind.

2. Camera Systems in Games

Since the development of the first 3D games, different ways of showing the action to the player have been experimented with. Some types of camera systems seem to have been definitely abandoned, while many do not have a definite form and keep evolving from one game to another. Below we will review the state of the arts in game camera systems:

- ***Fixed Camera Views***

To track the character, some games use a set of predefined cameras whose position and orientation have been set during the making of the game. The concept of multiple cameras is simply a convenient metaphor for the game developer. Indeed, during the development, predefining a view consists in positioning a camera object in the desired spot. The concept is equivalent to instantly moving one camera to a new location. The camera may rotate or pan but always within a predefined range. When the character goes off camera or is too far from it, the system switches to a new camera. The main advantage of this technique is that it allows precisely describing each scene by showing the player what he is supposed to see under an angle that is coherent with the story. In Cozic (2003), I describe in details the first encounter between the player character and one of the creatures in *Resident Evil 2* (1997).

Fig. II.2 (Shot I, II, III, IV). Capcom: *Resident Evil 2*, 1997. A series of shots leading to the first encounter between the main character and one of the creatures.

The encounter is prepared by a series of shots carefully chosen that aim at creating suspense. First, as the player moves towards the place where the creature is located, a shot shows a window that looks onto the outside. Through it the player can briefly see an undefined form passing by (Fig. II.2, Shot I). Once the player has opened the door (in Shot I), the next shot shows the character through a window in a subjective view (as being through the eyes of the creature). The window is at a few centimetres from the character (Shot II). As it is made of wood and can obviously be broken easily, the shot highlights the dangerous situation the character is in. Finally, the character passes a dead policeman lying on the floor (Shot III). The shot IV just before the encounter shows some blood drops falling from the ceiling and forming a puddle on the floor in the foreground. This series of shot hold the suspense until the actual encounter and shows that predefined viewpoints can be used to create a dramatic impact in games without the use of cut scenes. The game designer, conceived as a kind of filmmaker, has the possibility with this system to select meaningful shots that are both expressive in terms of affect and information. In *Alone in the Dark* (1992), the developers also freely use this possibility in the game introduction to set up the scene and the game mood (Fig. II.3).

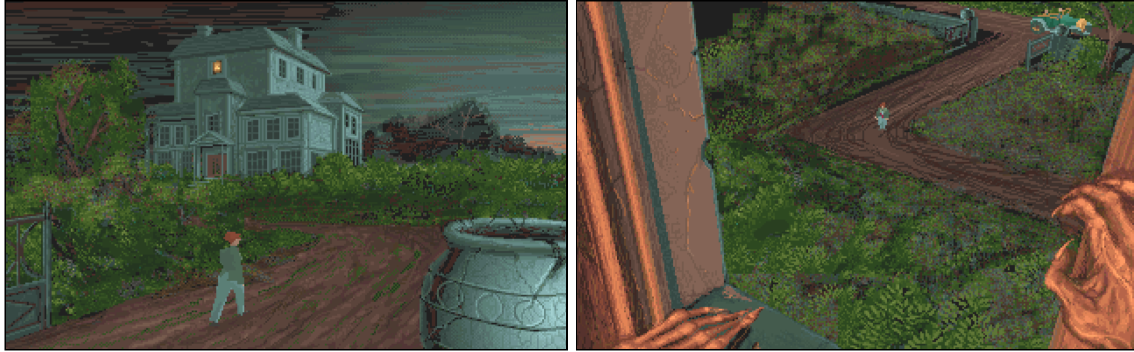

Fig. II.3. Infogrames: Alone in the Dark, 1992. Left: Derceto where the action will take place. Right: View showing the character from within the house.

However, it is often difficult with this technique to make views that are both playable and dramatically relevant. In this kind of system, a viewpoint should not hinder the player's actions because, as he does not control the camera, he will not be able to choose a better one. The risk is of making the game artificially difficult by masking a hazard he should have seen. Thus, the narrow shots of *Resident Evil* sometimes do not show the player the enemies that can nevertheless be very close; thus forcing the player to shoot randomly in every direction.

- ***Fully Interactive Camera Systems***

Another solution found in some games is to give the player complete control over the six degrees of freedom of the camera. It has been done in *The Legend of Zelda: The Wind Waker* and *Super Mario Sunshine* where the player can rotate and dolly the camera using a small joystick. This could be considered as a universal camera system as it could theoretically fit any situation. Indeed the player can always translate and rotate the camera to find a better view. However, I think that this solution should be avoided because it forces the player to consider too many parameters at the same time which can lead him to lose concentration on the game-play itself. Besides the task of controlling the camera is arguably not as entertaining as controlling the character.

- ***Tracking Camera and Semi-interactive Camera Systems***

At the most simple level, the tracking camera is the one that follows the character from behind and points at the same direction as him. However this can make the game particularly tricky in some situations where the player needs a greater sense of depth or needs to estimate distances – for example to jump from one platform to another. In *Crash Bandicoot* (1996), jumps between platforms are often haphazardly as the player may have difficulties to evaluate the distance between platforms in the view from behind (Fig. II.4). As the camera angle is always the same, it can also make the game monotonous.

Fig. II.4. Naughty Dog: *Crash Bandicoot*, 1996. The view from behind makes it difficult to evaluate the distance between platforms.

In order to deal with these issues, some systems offer in addition to the default following mode the possibility for the player to dolly or rotate the camera around the character within a predefined range. This type of partly user-controllable camera system solves some of the problems noticed for the fixed camera views. Especially here if the default view does not suit the player, he has the possibility to change it for a better one. It is also more user friendly than the fully interactive systems described above as the possible locations of the camera are usually limited to the most useful ones. For example in *Super Mario 64*, each press of the rotation button will rotate the camera by 45 degrees (Fig. II.5). That way one press of a button is usually enough to go from an uncomfortable view from behind to a side view without having to deal with the six degrees of freedom of the camera.

Fig. II.5. Nintendo: *Super Mario 64*, 1996. In terms of gameplay, a simple 45 degree rotation of the camera is often enough for the player to make better sense of his avatar's surrounding.

Although giving a lot of freedom to the player, this type of camera has the disadvantage of being difficult to control in narrow spaces. It tends to "knock" against the walls or to be obstructed by objects of the scene and, finally, it may not allow seeing the scene with as much efficiency as with predefined views. Currently none of these systems would be able to reproduce automatically the *Resident Evil 2* scene described above because the camera is only driven by the player's movement and never by the plot or by the characters' emotions. As a result this type of camera system only provides dramatically neutral viewpoints.

3. Current Issues

Through the previous reviews, two main issues have been outlined: on one hand the structure of these camera systems does not allow achieving good shot composition, and on the other hand they encounter difficulties when trying to find a balance between gameplay and cinematographic viewpoints.

Firstly, the scripts (or "idioms") can only be a partial solution to automated cinematography. They can be used to compute quickly a shot or a sequence of shots for specific situations such as a conversation between two characters, a character opening a door, etc. but they do not scale well to more complex virtual worlds. This is because an idiom has to make assumption on the relative positions of the objects: for example in a two character conversation, each character is expected to face each other and to be relatively close to each other; when the character opens a door, he is expected to be facing the door and so on. However this principle cannot be applied to complex scenes where the number of possible character / object configurations is infinite – such as a scene where a character is walking randomly in a forest or in a crowd. As a result, in this kind of situation most camera systems, including the idiom-based ones, will fail to achieve good composition or to show the player what he most needs to see to progress in the game (typically, the camera will be right behind the character, pointing at the same direction as the character).

One of the possible reasons is that the camera systems only consider the geometric data of the 3D world to compose the shot. These geometric data however only carry a very limited meaning. For example, the only thing that can be said using it is that a tree is bigger and geometrically more complex than a treasure chest, but there will be no way to know which one is the most interesting for the player. The fact that the camera system has an insufficient knowledge of the virtual world prevents it from achieving good composition. As seen earlier, it happens for example in Halper system: when the character enters the room in the medieval demo, the camera does not show the inside of the room.

Another problem is that the shots generated through the idioms lack diversity. Indeed currently this type of system processes each new scene by matching it to a database of scene templates and by outputting the camera parameters based on the template. The problem is that two rather different scenes may be categorized as belonging to the same scene template and thus the same camera parameters will be generated. Katz (1991) describes around ten different ways to film a conversation between two characters. Thus two characters in conflict are filmed under significantly different angles than two characters talking casually. However in an idiom-based systems both situations are likely to be assigned to the category “Two character conversations” and be filmed in the exact same way. Although it is possible that this problem could be solved by making a large database of scene templates that would cover any possible situation (in much the same way as some AIML chatterbots use database of millions of categories), we are looking for a more flexible approach with a system that could adapt to situations that have not been forecast.

The second main issue is that existing systems have not fully addressed the problem of finding a balance between gameplay and cinematic viewpoints. Existing camera systems usually only deal with one of these problems instead of conciliating both. Hornung *et al.* and Friedman *et al.*'s systems only deal with non-interactive stories; some others only handle specific situations via the use of idioms such as the systems developed by Amerson *et al.* and He *et al.* (at the moment they have been used mainly for dialogue scenes). Halper *et al.*, Hawkins and Giors' systems allow the same degree of interactivity as in a real game but do not attempt to make dramaturgically relevant viewpoints.

Finally games such as *Resident Evil* conciliate playability and cinematographic viewpoints but with the drawbacks described above: the camera system often hinders the player’s actions by not showing him what he needs to see to be able to play; it can create disturbing continuity problems; and it always shows the same scene using the same shots. It is also a system that could not be used in some games: for example those with unpredictable virtual worlds (such as an online role playing game) or those with such a large virtual world that it would be impossible to entirely cover it with predefined camera shots (such as the 3D versions of *The Legend of Zelda*). Finally, today many games, such as *Morrowind* (2002) or *Neverwinter Nights* (2002), can be modified by the addition of user-created plugins. The user can easily add new objects and constructions to the existing game virtual world. In these cases, the camera systems with predefined views will also lack flexibility: indeed a plugin could place an object or a building that could invalidate all the surrounding pre-calculated viewpoints.

III. Camera System Description

1. Proposed Camera System

- *Specifications and Requirements*

I have outlined in the rationale the overall objectives of the camera system and shown to what extent other people's systems achieve those objectives. From this enquiry, I deduced the following camera system specification and its matching requirements:

Specification	Example	Matching requirements
I. To show the character whilst avoiding occlusion.	To move the camera to the left or the right when the view becomes occluded by a tree.	To use ray-casting techniques or a "projective shadow casting algorithm" (Halper *et al.*) to avoid occlusion.
II. To show the relevant parts of the current scene.	To make an establishing shot when a new scene starts. To show the interior of the room the player has entered.	To make the camera system aware of what is important in the virtual world.
III. To make sure the player cannot lose the game because of an ill-positioned camera.	To focus on the hazards which are close to the player character.	To give the objects a priority and to be able to alter it depending on the context (for example an enemy close to the player will have higher chances to be included into the shot than an enemy that is far away).
IV. To do both specification I and II in a way related to the narrative.	To show a character that is meant to be dangerous or threatening with a low angle shot.	To have the camera to be driven by the narrative (This requirement requires the implementation of II and III)
V. To consistently handle the camera behaviour when a new event occurs or a new scene starts.	When the player character starts a conversation, to dolly the camera back in order to show both characters in the same shot.	To consider the context when creating a new shot (i.e. what are the previous shots, how long did they last, etc.) in order to be able to adapt to unexpected scenes or events.
VI. When the camera is moving, to handle its collisions with the virtual world.	To have the camera to slide along the wall when it is about to go through it.	To use ray-casting techniques to detect when the camera is about to collide.

Table 3: Camera system specification and requirements.

Most of these requirements are dealt with – at some level – by my system. However, I have decided not to focus on the collision and occlusion detection, which are already well handled by today's camera systems. It would be possible to combine the strengths of my system with these standard techniques. I have chosen instead to focus on the requirements IV and V, which are detailed below:

- *Requirement IV: Narrative Driven Camera System*

The camera system should have the ability to answer narrative related questions such as "Does the player have to take this object to progress in the game?" or "Is this non-playable character important with regard to the plot", etc. That way it will know what is important enough in the current scene, which in turn will help to select which objects should appear in each shot in order to achieve more relevant shot compositions.

- *Requirement V: Adaptability to Unexpected Scenes or Events*

The system should be kept as flexible as possible to be able to adapt smoothly to unexpected situations. For that reason the system should not use a database of possible situations (and their corresponding sets of shots) as in idiom-based approaches. Instead it would be preferable to use as generic as possible cinematographic rules which will be only expressed in terms of *Events*, *Objects* and *Player character* (In other words no knowledge about specific situations will be encoded). An Event is anything new that may happen in the current scene and an Object refers to a non-playable character or to an actual physical object. While designing the structure of this system, one of our expectations was that behaviours similar to those of idiom-based camera systems will naturally emerge from the interaction between a complete enough set of these generic rules.

A system that would include these specifications would need to have this kind of organisation (Fig. III.1):

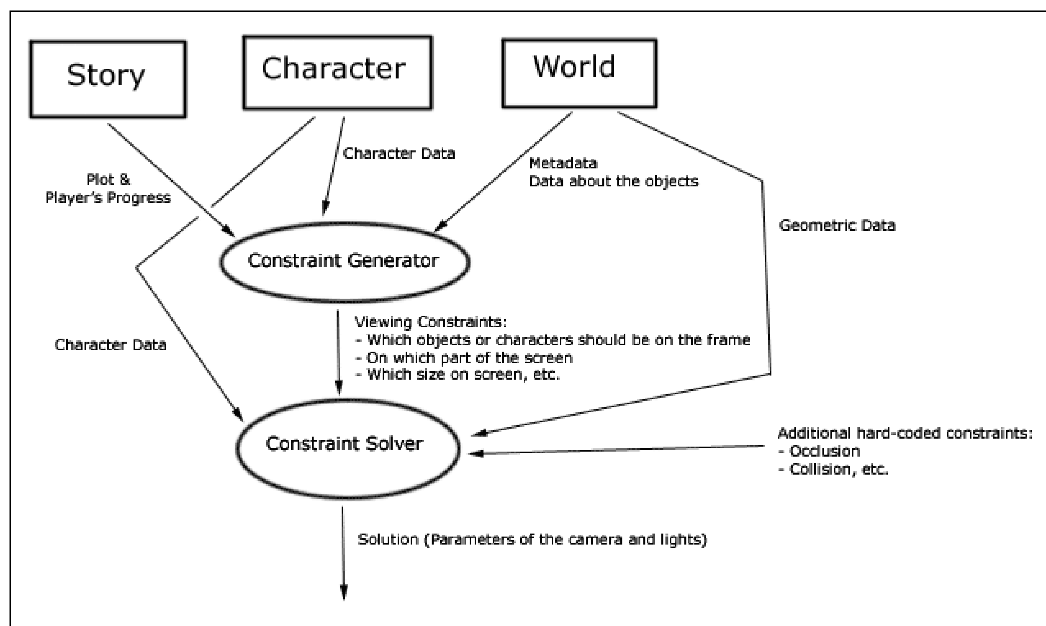

Fig. III.1. Camera System structure.

Story, *Character* and *World* are objects which provide in real-time the camera system with information about their current state. In my system, the *Story* module sends events to the camera system when

something new happens in the scene, it can also provide information regarding the player's progress (what he has already seen or done). The *Character* module gives information such as the position and direction of the player character, while the *World* module provides the geometric data and metadata of the object in the virtual world. Both data and metadata can be static or dynamic.

The constraint generator aims at generating dynamically the visual constraints. In order for the narrative to have an influence on the generation of the shots, the Constraint Generator takes its input from the Story module. The latter will provide narrative information about the virtual world such as the plot and the progress of the player and will allow the camera system to generate the right kind of atmosphere. By having the constraints decided by the plot and the progress of the player, a scene that has already been seen will be shot in a new, different way if the plot has changed.

The Character module provides the position, the orientation, the motion, etc. of the character in the current scene. When generating the visual constraints, the World module only provides the camera system with the metadata of the objects. These are additional data about the objects which allow answering the above questions. Technically, the metadata can be for example a marker indicating the degree of importance of the object with regard to the plot. It can be set during the game development and latter be modified at runtime. By taking into account simultaneously the character data and the metadata of the virtual world, it is possible, at a simple level, to guess the player's intentions. If the player has moved inside a room, the system will know that the player probably wants to see the inside of the room and thus will move the camera inside of it. This is possible because the room will not be considered only as a set of 3D points.

Anticipation should also be made easier by allowing the camera system to know what the player is more likely to be interested in in the next few seconds. For example if the player is about to arrive at an intersection with something on the left and nothing on the right, the camera starts looking toward the left (Fig. III.2a).

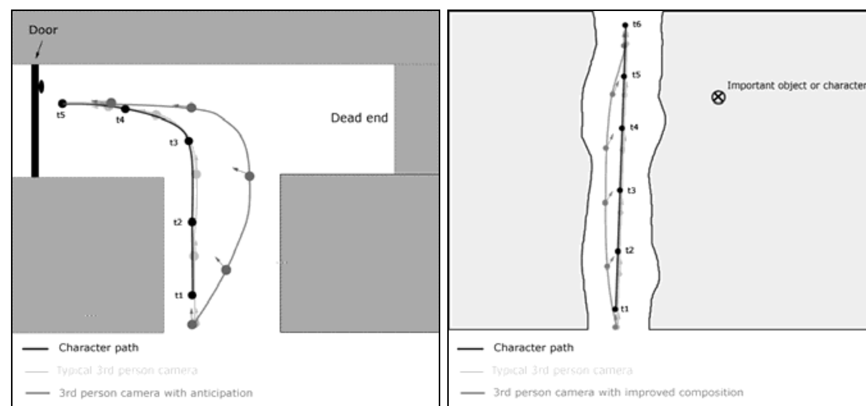

Fig. III.2 (a-b). Improved anticipation and composition.

Composition should also be improved as the camera system would be able to select the important objects that should be included into the shot (Fig. III.2b). By highlighting these important elements, the player will also be able to make sense of the virtual world more quickly. The importance of each object, named Story Contribution in my system, is described in more details below in the *Story Engine* section.

Once the constraints have been generated, they are sent to the Constraint Solver which will attempt to find a visual solution. The constraint solver is similar to those of Drucker, Halper or Bares and uses the geometric data of the World module as well as the character information to find a solution. Additional

constraints that should always be solved in all cases are also sent to the constraint solver (for example, the Occlusion constraint on the main character or the Collision constraint on the camera).

Once the shot to display has been selected, the system can use two different types of transitions. On one hand, if the old position and rotation of the camera are not too different from the new ones, the system will smoothly moves the camera from one spot to another. Otherwise it will instantaneously switch from one position to another, thus generating a cut.

2. Constraint Generator

The constraint generator analyzes the surrounding of the player's avatar and decides what is important enough to be into the next shot. This shot is then expressed in term of visual constraints which are sent to the constraint solver.

- ***Shot Contribution***

The constraint generator starts by finding out which objects are interesting for the player according for example to their importance with regard to the story. Once these important objects are found, the system decides how to compose the next shot according to cinematographic rules.

Each object is assigned a Shot Contribution (SC) value, which can be set up by a game designer and/or updated according to the plot. The value represents the probability of an object to be included in the next shot. The way the SC is assigned to each object depends on the type of game and on the game designers' goals. For example in a fast pace game, the game designers may just want to show the surrounding of the avatar in a way that will allow the player to make fast decisions. In this case, every object with which the player can interact could be given a high SC value in order to ensure that only "optimal views" are generated.

In an adventure game, where exploration is in itself a part of the enjoyment, the SC value of some object may be lowered in order to avoid on purpose some optimal views. In a game such as *Ocarina of Time* for example, the game designers may want to give the highest Shot Contribution value to Link's enemies during the fight scenes, and a lower or null value to some hidden places or even important characters so that the player still has to find them by himself.

Finally, the probability of the object to be in the next shot depends on the number of times (and for how long) it has already been seen in the previous shots. Roughly, as soon as it is assumed that the player knows where the object is, the constraint generator will be less likely to include it into the following shots.

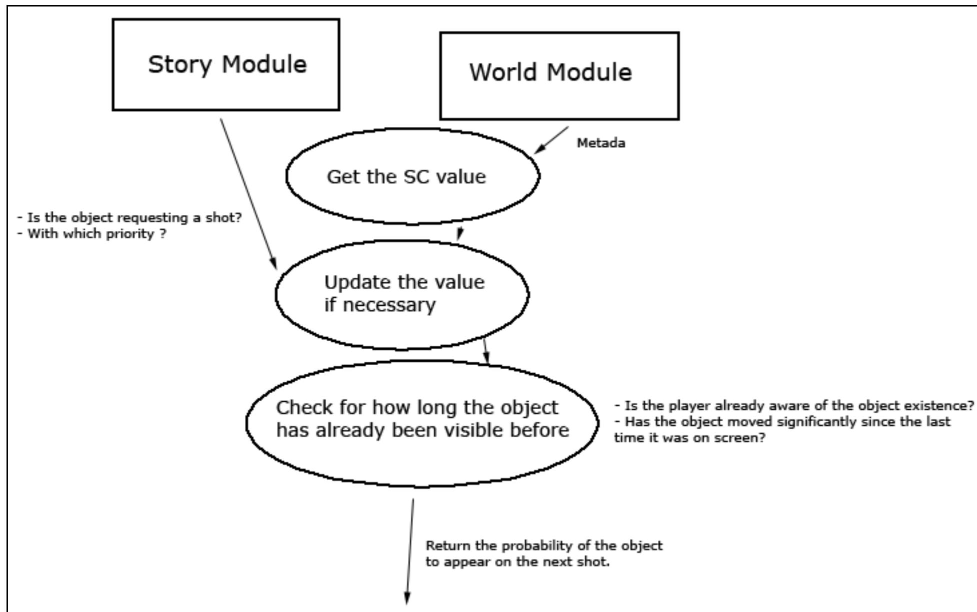

Fig. III.3. The complete process to calculate the probability of an object to appear in the next shot.

• *Rules for Shot Selection*

Once the system has selected the objects, many different shots may be possible so some rules are used to find the best possible one. As mentioned above, the system uses atomic rules which express simple ideas (as opposed to the more complex idioms described by He et al.). However complex behaviours can emerge from the combination of these rules (Fig. III.4).

(1) A shot should not last for "too long" or "too short" a time.
(2) If something new happens in the current scene, the viewer should be informed.
(3) Avoid jump cut.
(4) Do not cross the 180° line during an edit.
(5) The character should not look directly at the camera
(6) If the subjects are in an adversarial relationship, prefer the use of low-angle shots
(7) If the subjects are emotionally involved, prefer the use of close-up

Fig. III.4. Some example of generic rules.

The rules can be guessed by going through the idioms and finding out what are the underlying rules used by each line of each idiom. In the dialogue idiom decomposition below, the rules came out by asking for example "why do we need reaction shots?" (Shot I) or "why do we need a long shot of the actors when a dialogue starts?" (Shot II). Within an idiom-based system, two rules would be necessary to generate Shot I and II such as:

- If character A stops talking, show the expression of character B;
- If a dialog starts, display a long shot of the two characters.

The problem of these rules is that they only apply to very specific situations. It is the reason why I tried to use generic rules that are not tied to anything specific. As demonstrated in the dialog example below, the rules (1) and (2) combined with the Shot Contribution system are enough to generates Shots A and B. Beside, because these rules are generic, they could also apply to other – possibly very different – situations.

Decomposing in such a way the idioms can yield interesting results. As the rules are affected by dynamic information (such as the objects' metadata), two similar scenes will always be filmed into different ways. For example a conversation between two characters will be affected by the relationship between them according to the rules (6) and (7).

The fact that the system uses small building blocks also allows a greater degree of customisation. The camera system could be taught to prefer camera movements to cuts, to privilege low camera angles or wide shots, or to ignore the "180º rule". That way a game designer could give the game its own cinematographic style.

To illustrate these ideas, the following example takes an idiom (in bold) from the *Virtual Cinematographer* and shows how the Shot Composition value and the atomic rules will automatically recreate the idiom (in italic):

- When the dialogue scene starts, make a two-shot including the two actors.

When a character initiates a dialog (in a game it is usually done by pressing the "action" button when being near another character), the two characters update their SC to the maximum (in this example, it is assumed that the object with which the player wants to interact will automatically have a high SC value, as this object is logically the player's centre of interest). As it makes a significant difference with their previous SC values, the system detects a new scene and therefore triggers the rule (2). A shot of the two characters is generated.
As the player has now seen the two characters and has an idea of their spatial relationship to each other, their SC values starts to decrease.

- When A talks, make a shot of A

If A talks, it instantaneously increases its SC. As a result it becomes higher than B's (which is still decreasing), and thus triggers again the rule (2) with a long shot or a close-up of A. The selection of the shot size may depend on additional rules such as (6) or (7) for example.

- When B talks, make a shot of B

(Ditto)

- If an actor has been in the same shot for more than two seconds, get a reaction shot from the other actor.

Here the atomic rules should make the shot selection both more unpredictable and more consistent. On one hand, the system, via the rule (1) (A shot should not last for "too long" or "too short" a time), will check on each iteration for how long the current shot has lasted and generate a new one if necessary. If this happens, the other actor is likely to be included in the next shot as his SC value is still relatively high, thus automatically generating a reaction shot.
The moment the system switches to a reaction shot should also be more consistent thanks to the rule (2). Indeed it will make the system switch the view to the other actor whenever something new happens with him (and not only after an arbitrarily set time). For example, if one of the characters suddenly changes its mood because of what the other said, its SC value will change significantly and thus a reaction shot will automatically be generated via the rule (2).

- ***Requesting a Shot***

As in Tomlinson's system, the possibility to request a shot could easily be implemented by allowing the objects (via the game editor) to temporarily update their own metadata. For example, a fisherman sitting quietly by the river would have little importance in term of game play or with regard to the story. Now if he suddenly falls into the water, his importance would raise accordingly (for example because now the player can choose to rescue him). In this case if the avatar is close enough, the camera will automatically compose a new shot including the fisherman (whether by moving the camera or by quickly inserting a long shot of the scene) in the same way a viewer would look in the direction of the splash noise.

Finally the constraint generator sends the visual constraints (the list of objects and the shot size) to the constraint solver.

3. Constraint Solver

The aim of the constraint solver is to set the camera parameters given the requested shot and the geometric and visual constraints. It iterates through all the rules and tries to find a single shot that would satisfy all of them. If no solution can be found, it iterates a second time (and up to eighteen times in the current system) with relaxed parameters. The system being object-oriented, the rules are designed to be self-contained and thus to be able to relax their own constraints depending on the number of iterations already done. For example, the "Object visibility" rule aims at keeping as many of the objects requested by the constraint generator as possible in view. However if no solution showing all of them can be found during the first iteration, the rule will ignore the less important objects during the following ones.

IV. Camera System Development

The last chapter of this thesis will describe how the camera system has been implemented.

1. Software Used

Below is reviewed the main software programs used to develop the camera systems and to test it.

- ***Discreet 3D Studio Max 6.0***

3D Studio Max is a 3D modelling and animation tool, which comes with features made to simplify game design, including low-polygon modelling tools and character animation tools, and a scripting language. It also supports exporting to Shockwave 3D format, a convenient and small format used by Adobe Director. In this project 3D Studio Max has been used to create the 3D virtual world as well as to author the game story and the interaction between the objects via *The Intruder* game editor.

- ***Adobe Director***

Adobe Director is an authoring tool for creating multimedia interactive content. Its recent versions include a 3D module which supports all the necessary features to make simple games or any kind of 3D interactive virtual worlds. It is easy-to-use thanks to Lingo scripting language and can be used to import W3D file from 3D Studio Max thus allowing rapid prototyping of our camera system.

- ***AIML***

The knowledge base of my system is based on AIML, which stands for "Artificial Intelligence Markup Language". It is a programming language developed by Dr. Richard Wallace and used to design natural language software agents. Its structure includes several elements among which the *categories*, *patterns* and *template* are the most important. The Category is the fundamental unit of knowledge and it is composed of Patterns and Templates. The Pattern element contains the pattern that will be searched for within the user's input. If it matches it, the Template will output the answer of the agent. This simple logic can also be used in a similar way to write rules of cinematography. In my system, the knowledge base rules request the current state of the virtual world (such as the position of the characters or the direction of the player avatar). If this state matches the rule statements contained in the Condition element (equivalent to the Pattern element in AIML), it will run the additional statements contained in the Action element (equivalent to the Template element). Please see *Camera System: Knowledge Base* below for more details on the knowledge base.

- ***Choice of the game engine***

Different game engines have been considered to implement the camera system. *Neverwinter Nights* provides a game editor to create new stories. It allows editing any aspect of the game, whether it is the plot, the graphic or the interactions. It also has a large collection of online mods, which would have been very convenient to easily test the camera system in various scenarii. However the game editor only allows

a very limited control of the camera. It is locked on the main character and can only rotate or zoom around him. The fact that the character always has to be in the centre of the screen makes it unusable to develop our system as one of our assumption is that the player's centre of interest is not only his avatar but a composition of his surrounding.

At some point, the *Renderware* game engine, which has been developed by Electronic Arts and Criterion Software, has also been considered to develop the camera system. It is a complete tool, which includes everything that may be needed by a game developer. However it has been decided not to use it as it would have been necessary to develop a whole game prototype before being able to start developing the actual camera system. Learning to use the engine may also have been too time-consuming.

The Intruder game engine appeared to be a good compromise between the easy game editing capabilities of *Neverwinter Nights* and the flexibility of *RenderWare*, without the cost of having to learn a new system. I developed this engine for a game adventure demo originally intended for an MA (Cf. *The Intruder - Expressive Cinematography in Videogames* in appendix). It is developed using Lingo object oriented programming and is organized in a hierarchical structure which made it relatively easy to integrate the new camera system. It also includes an event based engine: any object can received a message (such as *#userAction*, *#pushed*, etc.) and react using the script that is attached to them. This system allows the creation of a simple story line that is suitable to test the camera system in various situations.

• *The Intruder game editor*

The game editor of *The Intruder* runs under 3D Studio Max and is developed using Max Script. 3D models and animations are designed using traditional Max tools and reaction scripts, interactions and metadata are assigned to the objects using the editor. Finally, the whole scene is exported as a set of XML files and W3D files. For example, in the demo, a character sets up to follow the player's avatar is made of two custom states: an #idle state, and a #followCharacter state. When the character is idle, he waits for the player avatar to be 400 units away from him (Fig. IV.1a). When it happens, it changes its state to #followCharacter (Fig. IV.1b). This state automatically sets the Story Contribution value (described below in the *Story Engine* part) of the character to 1.0 so that all objects are notified that a narrative event happened. The character will then start following the avatar. Below are the two Lingo scripts used in 3D Studio Max.

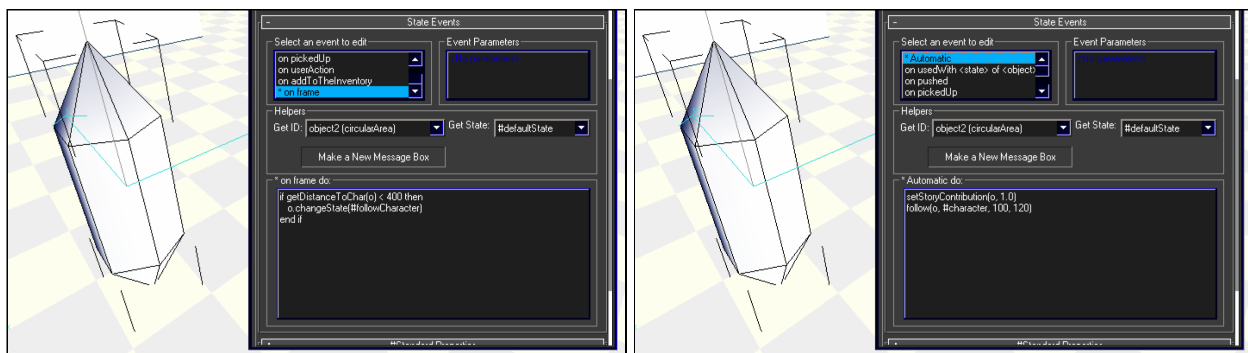

Fig. IV.1 (a-b): the #idle (a) and #followCharacter (b) states of an object sets up to follow the avatar, in 3D Studio Max.

- ***Editing process***

The diagram below shows the editing process and the various file formats used by the game engine, as well as the software used to create them:

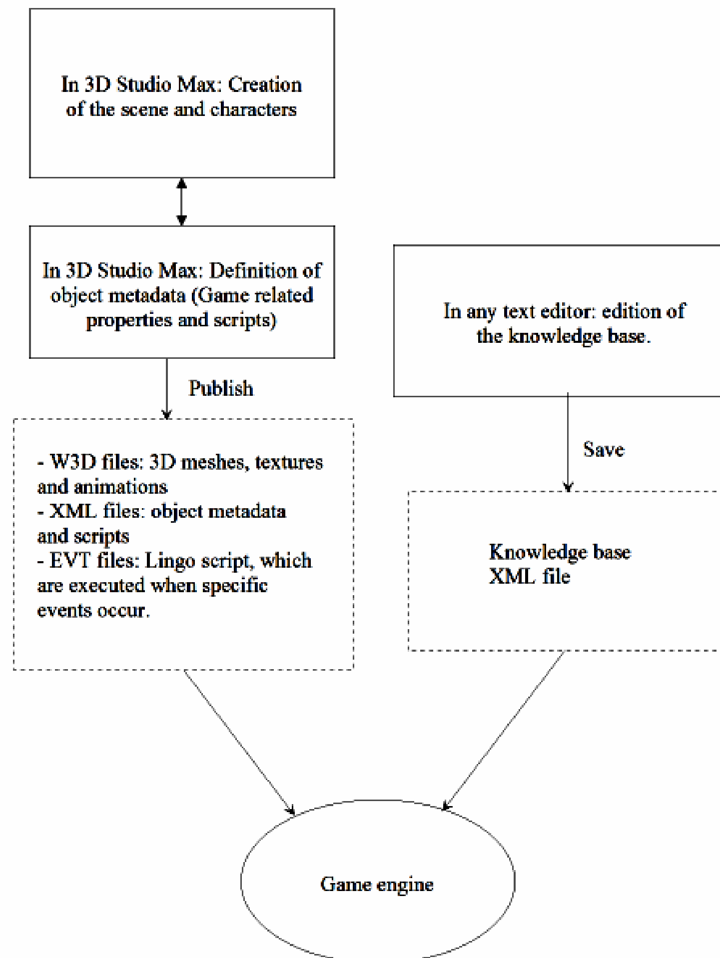

Fig. IV.2: Diagram showing the editing process.

2. Story Engine

The story engine has been added to the structure of *The Intruder* in order to be able to fully test the camera system. It consists of a single class that gathers and computes narrative information about the virtual world and passes it to other objects when necessary.

In our case the story engine has two main functions. On one hand it is used to set the narrative importance of each object. Via the game editor, a *storyContribution* value is assigned to each object so that the story engine is able to know which object is important with regard to the plot. Its value goes from 0.0 to 1.0 (the sum of all the SC values in the scene does not need to add up to 1) and has a meaning that depends on the type of game. For example in *The Intruder*, which is an adventure game demo, the storyContribution

value has three thresholds – 0 is used for objects or characters that belong to the décor such as a painting on the wall or a passer-by. Not noticing these objects will not have an impact on the storyline. On the other hand, objects with a 1.0 are essential for the progress of the plot and cannot be ignored – it can be a character to whom the player has to talk to or a house to which the player has to go. Although it is not currently used, an optional 0.5 storyContribution value could also be assigned to other important objects. It can be used for example for a character that gives the player a sub quest to complete or an object that gives information on the story background – in general anything that can be interesting in terms of gameplay. Whether the player notices these objects or not however will not have an influence on the main plot. The storyContribution value can also be changed dynamically during the game in order to reflect the plot progress. Thus, a character whose role is to give an important object to the player's avatar will have a 1.0 value but once it gives the object, its value will be set back to 0.0. The story engine notifies the camera system of any similar change in the story so that it can react by computing new shots if necessary. For the same reason, it also informs the camera system whenever an important object is near the player's avatar.

The following table describes the story engine class:

cStoryManager PROPERTIES				
Name	**Type**	**Default**	**Example**	**Description**
Scene	Instance	N/A	N/A	A reference to the current scene and its objects.
Camera System	Instance	N/A	N/A	A reference to the camera system to which the story engine can send messages when a new event occurs.
Character	Instance	N/A	N/A	An instance of the player character.

METHODS			
Name	**Input**	**Output**	**Description**
New	Scene	A reference to the new object.	Initializes the story engine and assigns it the scene it is going to monitor.
Set Story Contribution	Object, State ID, Story Contribution	N/A	Sets or updates the story contribution value for the given state of the given object.
Update	N/A	N/A	Updates the internal state of the story engine. It checks if anything has changed since the last update and if so notifies the relevant objects.

Table 4: Story engine class.

3. Camera System

• *Constraint Solver Implementation*

Drucker and Zeltzer (1994) argue that it is more efficient to express the characteristic of a shot in terms of visual constraints than using the six degrees of freedom of the camera. This indeed would be the equivalent of a film maker positioning his camera using a story board. In order to implement and test these type of visual constraints for my system, I developed a small program (Fig. IV.2). A set of functions and a small interface has been build in order to test them with various parameters. The three functions allow me to express a shot in terms of:

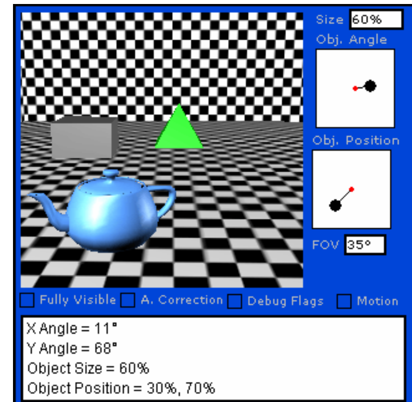

Fig. IV.2. Constraint solver interface.

- *the size on screen of the objects*. The function takes as an input the percentage of the vertical screen space a given object should fill. It then outputs how far away from the character the camera should be for it to happen.

- *the angle under which the object should be shown.* It takes as an input the angle between the screen plane and a character, and outputs the corresponding rotation for the camera.

- the position on screen of the object. It takes as an input the X and Y position (expressed as a percentage) that the object should have on screen and outputs the position for the camera.

The three functions demonstrated in this small program are:

METHODS			
Name	**Input**	**Output**	**Description**
ModelSizeToCameraDistance	Camera, Model, Percentage	Distance between the camera and the model.	Given a camera and 3D object, this function returns the distance between the object and the camera so that the object appears to take the given percentage of the total screen space.
ModelAngleToCameraRotation	Camera, Model, RotationX, RotationY	Camera rotation angle.	RotationX and RotationY are the angle between the screen plane and the 3D object. Given these parameters, the function outputs the corresponding rotation of the camera.
ModelPositionToCameraPosition	Camera, Model, PositionX, PositionY	Position of the camera.	PositionX and PositionY define the position on screen of the given object. From these parameters, the function outputs the corresponding position of the camera.

• *Knowledge Base*

The aim of the camera system is to generate in real time the best possible shot for the current situation. As described above it is composed of two main parts: the *constraint generator*, which generates a list of possible shots for the scene; and the *constraint solver*, which selects the best shot among them and outputs the parameters for the camera. Each part uses its own set of rules grouped in a knowledge base.

- *Camera System Rules*

The constraint generator and the constraint solver each use a different set of rules. As one of the aims of this camera system is to conciliate dramatic camera views and playability, the rules derive both from cinematographic knowledge and from gameplay considerations. They are encoded as an external XML file, the structure of which is inspired by AIML. This structure of this language can conveniently be used to model my knowledge base. As described below, the elements have been renamed but their function is similar:

- *Knowledge Base Structure*

```
<XML version="1.0">

<INITIALIZATION>
   (Lingo script)
</INITIALIZATION>

<RULE name="Rule Name" appliesTo="(Objects or Shots)">

    <CONDITION>
        (Lingo script)
    </CONDITION>

    <ACTION>
        (Lingo script)
    </ACTION>

</RULE>

<RULE name="Another Rule"...

etc...

</XML>
```

Fig. IV.3. Knowledge base structure

As for AIML, there are three main elements: the *Rule* is the fundamental unit of knowledge. It includes a *Condition*, which checks if the rule is triggered by the current scene. If it is, the optional *Action* part is executed. There are two different types of rules, those that are used by the constraint generator and that apply to objects and those that are used by the constraint solver and that apply to shots. The first type of rule helps to find out which shots would be ideal for the current situation, and the second one checks if it is indeed possible to generate such shots within given constraints (such as gameplay constraints, visibility constraints, etc.). Finally an *Initialization* element has been added – here can be put anything that needs to be initialized before the camera system starts. For example, the field of view of the camera or the minimum shot duration.

The Condition, Action and Initialization tags include a Lingo script which is imported and compiled at runtime by the game engine. Within these tags, it is possible to access properties and methods of the camera system via the predefined variable “CS”. It is also possible to access the functions that were primarily built for the game editor. This includes functions to create dynamically global variables of the game engine, which can then allow the different rules to communicate between them. In our knowledge

base, this is used for example by the “character visibility” rule to tell the other rules whether the character is currently visible or not.

The Condition script always returns a value which tells whether or not the rule applies to the current scene. It is only when it returns “true” that the Action script is executed. The Action script for the Object rules always contains statements to generate and queue a new shot.

Each rule used by the system is described in detail in *VIII. Source Code: Knowledge Base*.

- ***Camera System Integration and Generic Entity***

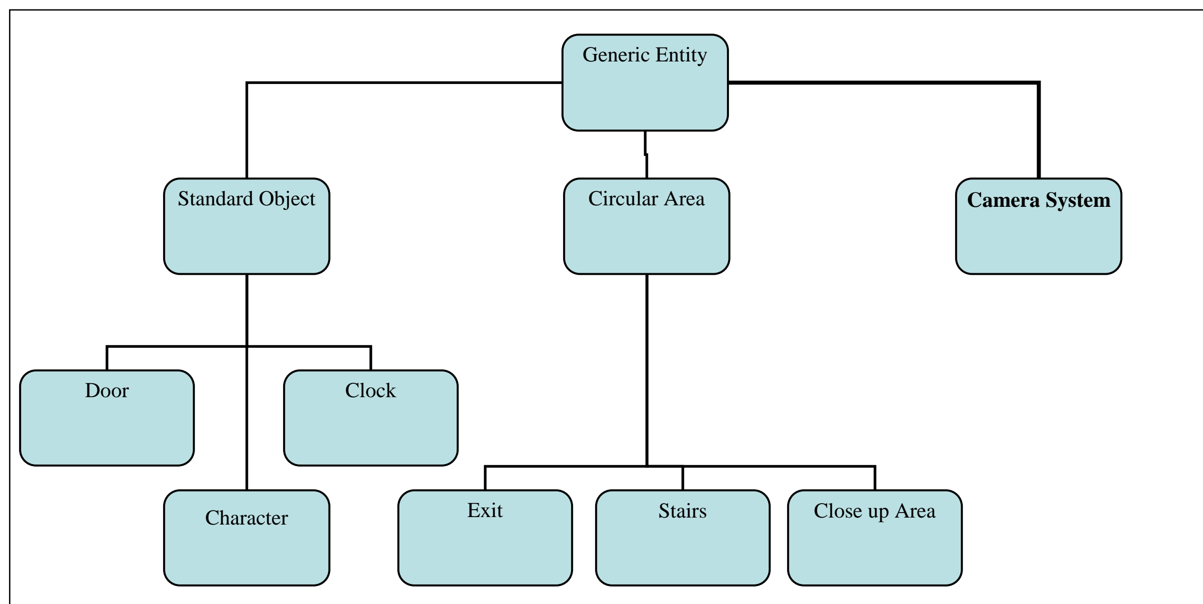

Fig. IV.4: OOP class inheritance diagram of the game engine. The camera system class inherits from the Generic Entity class.

The camera system inherits from the generic entity class (Fig. IV.4). It is the basic class of the camera engine and it encapsulates all the low level game mechanisms. This includes especially the object state management: any object can be set to have a certain number of possible states and the generic entity class will create and set all the properties assigned to each state as well as the transition between two states. Additionally the class handles the messages sent to the object by the player or by other objects. Thus the camera system receives messages from the story engine when something new happens in the scene. The state mechanism is not used by the camera system but would allow flexibility in the case we would like to extend it. For example using two different states, we could create a specific camera behaviour for the external scenes and another one for the interior scenes (Cf. *The Intruder - Expressive Cinematography in Videogames* for a detailed description of the generic entity class).

- ***Camera System Implementation***

The camera system is made of two main parts: the constraint generator and the constraint solver. The first one aims at generating a list of possible shots for the current scene. A large part of this work is done via the story engine (which maintains a list of relevant objects with regard to the story) and the Object rules in the knowledge base. The constraint solver aims at calculating the camera parameters from the current shot information and checks if the new camera position and orientation do not break any of the Shot rules.

Below are described the characteristics of a shot and the steps followed by the system when updating the camera parameters:

- *Character control*

The character's control in a 3D game can be implemented in two different ways:

- In one model, when the player presses the "up" button, the character moves towards the direction he is facing. Pressing the "left" or "right" buttons will rotate the character left or right. This is for example the case in the Resident Evil series.

- In the other model, when the player moves the joystick forward, the character moves *away from the camera* and not necessarily towards the direction he was facing. Some examples of games include *Super Mario 64* or *The Legend of Zelda: Ocarina of Time*.

The second type of character control is particularly suitable when the controller is analogue (i.e. a joystick). Usually, the speed of the character will depends on how far the joystick is pressed, which allows for precise controlling. The main disadvantage is that quick camera movements can make the game confusing. For example, if the player presses the joystick forward and if the camera quickly rotates by 90° around the character, the character will no longer move forward but will instead turn left. In some games, this issue is dealt with by letting the character move in the direction he was already facing until the player releases the joystick. When he/she moves it again, the character starts walking again away from the camera.

The first type of character control does not have this problem; however it also does not have the flexibility and precision that an analogue system would offer. For example, to move the character in a different direction, it is necessary to rotate him using the "left" or "right" button, wait until he is facing the required direction and then press "up" to move forward. On the other hand, with an analogue control, moving in another direction is done instantaneously by moving the joystick in the required direction.

In order to keep it simple, and to focus on the camera system rather than the character's control system, I decided to use the first model, where the player uses the keyboard arrows to move the character. However with some additional work on the control system, it would be possible to use an analogue stick with this camera system. It would be necessary to implement the aforementioned technique where the character keeps moving in the same direction for as long as the player keeps the joystick down, even when the camera angle changes.

- *Shot Characteristics*

A shot object can be thought of as a frame from a story board, as it holds all the information in terms of composition (and movement) of the objects over an interval of time. This information includes: the objects that should be visible (cf. "Shot queue and Shot selection" below), the size on screen of the objects, the time when the shot should start, the type of transition between this shot and the previous one, and a priority value.

Below is a complete description of the shot class:

Shot Properties			
Name	**Type**	**Default**	**Description**
ID	Integer	N/A	Shot unique ID

Objects	List	An empty List	List of objects to be included in the view
Priority	Float	0.0	From 0.0 to 1.0. The priority of the shot
Start Time	Integer	N/A	The time at which the shot is going to start
Can Change	Boolean	TRUE	A flag to indicate whether or not the shot can be changed. It can be used for example to specify that the shot cannot be changed if it has not lasted for the minimum shot duration.
PC Visible	Boolean	TRUE	A flag to indicate if the player character has to be visible in this shot. It can be useful if we want for example to make an insert shot of some object. If the shot does not last for too long, the player character does not necessarily have to be visible.
Transition	#cut or #move	#move	The type of transition between this shot and the previous one.
Shot Size	List	An empty list	A list containing the size of the objects as they should appear on screen.

Table 5: Shot class

- *Step 1: Shot Queue and Shot Selection*

The shot queue contains a list of the most relevant shots for the current scene and situation. Each shot is added to the list by the "Object" rules in the knowledge base, which also assign them a priority rate. The selection of shots is then straightforward; the camera system simply goes through the list and selects the shot with the highest priority rate. Once it is done the queue is discarded.

- *Step 2: Calculating the Camera Parameters*

The second step is to calculate the camera parameters from the shot information. The system ensures first that all the required objects are in view. This is achieved by adding to the virtual world an invisible circle that includes all the objects (Fig. IV.5). The camera then points at this object and is moved in such a way as to show the entire circle using the functions described above in "Visual Constraint Implementation".

At this stage, the generated view may be wrong: some objects may be occluded or the camera can be incorrectly positioned inside another object.

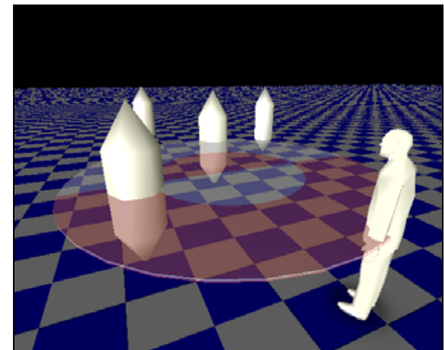

Fig. IV.5: as a first step, the camera is positioned in such a way as to show the circle that includes all the important objects.

- *Step 3: Checking the Shot Validity*

This step ensures that the previously generated view is valid using once again the knowledge base. The camera parameters are sent to the "Shot" rules, which check their validity and modify them if necessary. The decision made is then sent back to the camera system. The rule checking process is repeated as long as not all the rules return a positive result. On each new iteration the constraints are slightly relaxed (here this is done by removing from the shot the less important objects) in order to find an alternate solution. If after a number of times the constraints are still not satisfied, and in order not to slow down the game, the system will stop iterating and keep the last camera parameters. In this case, the view is likely not to include all the characters or to have some of them entirely occluded. However, the movement of the characters will usually naturally solve the problem on the next camera system update.

This step of checking the shot validity also manages to give a more dynamic feel to the game. By constantly trying to find a more suitable view, the system cuts from one shot to another or moves and rotates the camera around the characters, thus giving more pleasing and varied visuals (Fig. IV.6a-b). As the basic rules of cinematography are respected, the result is usually not disturbing.

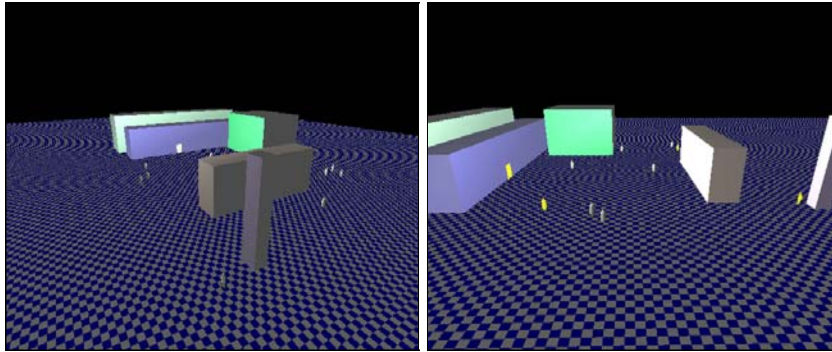

Fig. IV.6a-b: the constraint specification are the same in these two shots (which is to show in one shot all the important (yellow) objects in the scene), however the resulting camera parameters are significantly different. In the second case, the system has to compose a new view when the character moves behind the pillar.

- *Step 4: Deciding on the Type of Transition*

Currently the type of transition between two shots is chosen by comparing the previous camera parameters and the new camera parameters (generated by the previous step). If the distance between the two shots is not too important and if the angle between them is greater than 180 degrees (the rules are likely to prevent this case anyway) the camera will move. Otherwise the system will cut from one shot to another.

- *Step 5: Shot Contribution Update*

The shot contribution value of each object in the current shot is decreased on each update of the camera system. Once it reaches zero, the object is removed from the shot. This ensures that objects that have already been seen by the player do not keep appearing in subsequent shots (they can still appear in the view but the system will not attempt to compose a specific view for them). This behaviour comes from the assumption that an object that has been visible long enough becomes less important in terms of information: once the player knows its location, it is not necessary to show it anymore.

- *Step 6: Shot History*

The shot history simply keeps a list of the most recent shots (the last thirty shots) that have been selected by the camera system. It does not have a specific function inside the camera system but it can be used by the rules to generate new shots based on the previous shots. Thus in our system the “Establishing Shot” rule checks how many times each object has already been seen – if none has been, it considers that a new scene has just started and suggests the creation of an establishing shot (The way previous shots influence the choice of current shots is discussed in more detail on page 41, *Player’s Knowledge Consideration*).

- *Properties and Methods*

Below is the complete list of properties and methods used by the camera system. See also *VIII. Code Source: Camera System* for more details on the camera system implementation.

cCameraSystem PROPERTIES				
Name	**Type**	**Default**	**Example**	**Description**
Ancestor	Instance	N/A	N/A	A reference to the generic entity ancestor.
World	Shockwave 3D	N/A	N/A	A reference to the 3D virtual world
Player character	Instance	N/A	Character(“Victor”)	A reference to the player’s avatar
Camera	Camera	Camera(1)	Camera(1)	The 3D world primary camera
Shot Contribution List	List of Objects	[]	N/A	The list of important objects in the current scene and their associated shot

				contribution value.
Current Shot	Shot	Empty Shot	Cf. Shot Characteristics above	The current shot properties
Influence Area	Model	VOID	Model(“cylinder”)	If enabled the camera system will only consider the objects inside this area.
Max Iteration	Integer	18		The maximum number of iterations allowed to solve the constraints.
Iteration Count	Integer	0	5	The number of iterations since the beginning of the update
Model To Look At	Model	VOID	Model(“cs_modelToLookAt”)	Invisible model to which the camera points at.
Save Camera Rotation	Vector	VOID	Vector(70, 0, -120)	Camera rotation value at the beginning of the system update.
Knowledge Base	Property List	[#shotRules:[], #objectRules:[]]	[#shotRules: [<offspring "Object Visibility" 1 a31e8a8>, <offspring "Character Visibility" 1 a363cd4>], #objectRules: [<offspring "Establishing Shot" 1 2909118>]]	A reference to the knowledge base with the shot rules and the object rules in two different lists.
Shot History	List of Shots	[]	N/A	List of the shots that have recently been selected by the camera system.
Shot Queue	List of Shots	[]	N/A	List of possible shots for the current scene.
Start Transform Target Transform	Transform	VOID	N/A	Used to smoothly move the camera from the first transform to the second transform over a given duration
Maximum Shot Duration	Integer	8000	5000	Maximum shot duration in milliseconds
Minimum Shot Duration	Integer	1000	1500	Minimum shot duration in milliseconds

cCameraSystem METHODS			
Name	**Input**	**Output**	**Description**
New	3D Member, Camera ID	A reference to the new object.	Initializes the camera system and its properties.
Track Character	Character	VOID	Tells the camera to track the given character.
Set Influence Area	Character, Radius	VOID	Sets up the (optional) influence area of the camera system. The system will ignore all objects that are not in the circle defined by the given character position and radius.
Make Script	Script Name, Script Text	Script Reference	Creates a new script and returns an instance to it. Used to import the knowledge base scripts.
Import Rule Set	File Name	VOID	Imports the external knowledge base (an XML file) into the system.
New Shot	Objects, Priority, Character Visibility	Shot	Creates a new shot with the given objects and priority. Optionally it is possible to specify whether the player character has to be visible or not (by default it will be)
Get Camera	VOID	Camera	Returns an instance to the primary camera.
New Event	Object, Event Name, Parameters.		Notifies the camera system that a new event has happened in the current scene. The system will accordingly update its shot contribution list. This method is mainly used by the story engine.
Monitor Object	Object, State, Shot Contribution	VOID	Notifies the camera system that the given object is important in the current scene, and should be included in some of the next shots.
Compare Shots	Shot1, Shot2	Difference between the two shots.	Compares the two given shots and returns the difference as an integer value. The higher the value, the bigger the difference between the shots.
Print Shot	Shot	String	Returns as a string various properties of the given shot.
Update	VOID	VOID	The main method of the camera system. It attempts to find the best shot for the current scene.
Get Iteration Count	VOID	Integer	Returns the number of iterations since the beginning of the system update.
Get Max Iteration	VOID	Integer	Returns the maximum number of iterations.
Which Transition	Transform1, Transform2	#cut or #move	Given two camera transforms, finds out which type of transition should be applied between the two associated shots.
Remember Shot	Shot	VOID	Saves the shot in the history list.
Get Relevant Objects	VOID	List of Objects	Returns the shot contribution value list
Count Number of Time on	VOID	Integer	Using the history list, returns the number of time an object has

Screen			been on screen.
Queue Shot	Shot	VOID	Adds a shot to the shot queue.
Get Shot Queue	VOID	List of Shots	Returns a reference to the shot queue.

Table 6 and 7: Camera system class

• *Camera System Behaviour Analysis*

Below is analysed the behaviour of the camera system in different situations using the knowledge base provided in *VIII. Source Code: Knowledge Base*. I will outline the strengths and weaknesses of the camera system and I will put forward possible solutions to some of the problems encountered.

- *Test bed*

I built a small but relatively complex virtual world (Fig. IV.7a) in order to test the camera system in various situations. In the first part of the scene, two large objects (one very high and one very large) have been placed in order to assess the ability of the system to avoid occlusions. Characters (Fig. IV.7b) have also been positioned in non-trivial configurations in such a way that they can often be occluded by the surrounding buildings or by other characters. Some characters are categorized as “important” with regard to the plot and some not. One of the characters will follow the player’s avatar in order to check the ability of the camera to track two characters at the same time. Finally some corridors have been built in order to check how the system performs in narrow places.

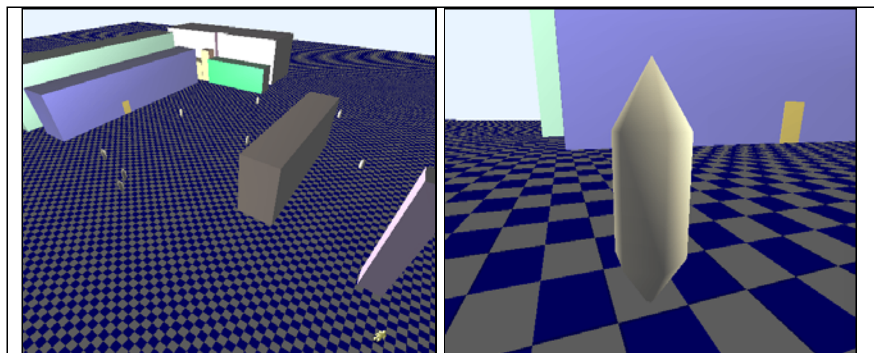

Fig. IV.7: test bed for the camera system (left). For testing purpose, this object represents a non-playable character (right).

- *Anticipation*

As expected the system allows the camerawork to better anticipate the player’s movements by showing the most important objects near him. In the example below, the character is approaching an intersection with nothing of importance on the left and a door on the right. In this situation, the camera slightly rotates in order to compose a shot showing the door and the character (Fig. IV.8a-c).

The feature however does not conflict with the demands of playability. Indeed if the player still wants to see what is on corner on the left, the camera system will generate the appropriate view (Fig. IV.8d) as soon as the character turns toward it (as is also described below in “Player’s knowledge consideration”)

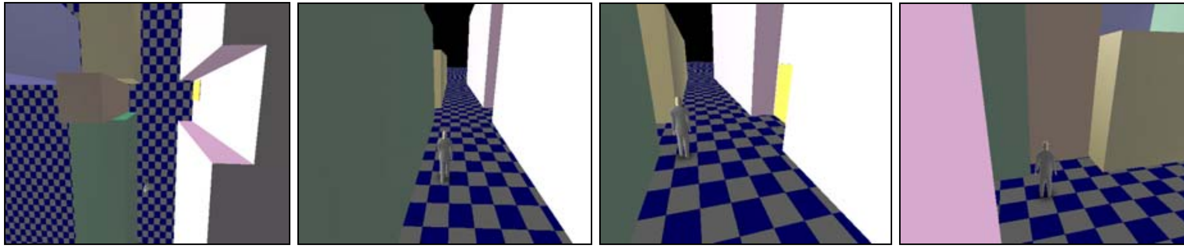

Fig. IV.8a-c: The camera slightly pans to the right to show the door. **IV.8d**: However, it still allows the player to see what he wants to see.

- *Plot Driven System*

The system also correctly reacts to the event messages that are sent to it by the story module. Whenever an event occurs in the virtual world the camera system generates a new shot including the event (if it is appropriate). For example, at some point a character is sets up to start following the player's avatar as soon as it moves nearby. The character changes its state from *#idle* to *#follow*, which is detected as a new event by the story engine, which in turn notifies the camera system:

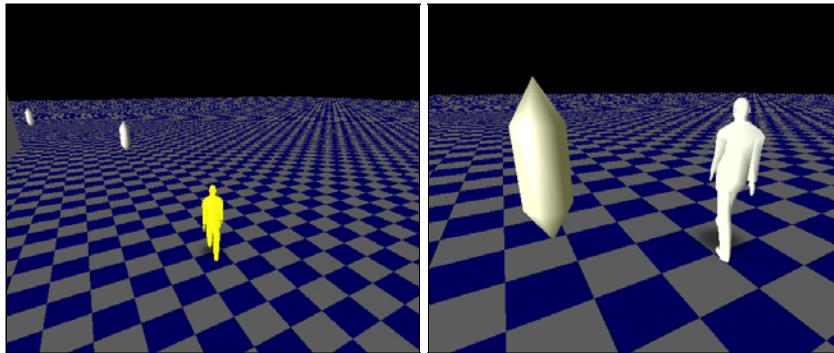

Fig. IV.9a-b: The change of state of the object is detected as a new event. The camera system is notified and therefore generates a shot of the object.

- *Player's knowledge consideration*

The system keeps an historic record of all the shots thus allowing the system to consider what the player has already seen to generate the shots. This feature is used in particular to avoid making useless shots that include objects that have already been seen by the player. When the "New Event Shot" rule is notified by the Story Engine that a new plot event occurred, it will first generate a temporary shot and compare it to the previous shots. If the shot is completely new it will be queued as usual, however if it is not, it will only be queued if the player is facing the scene. In other words, if the player character is turning his back to an already seen event, the system assumes that he is not interested in it for the time being and therefore does not queue the shot. This mean for example that if a non-playable character keeps trying to get the player's attention (by sending events via the story engine) and if for some reason the player decides to ignore it, the camera system will stop generating shots specifically for this character.

This feature was not there initially but it became necessary after the addition to the virtual world of an object that was following the main character everywhere. The camera kept on showing this object despite the fact that it had already been clearly shown the first time the player encountered it. However if the player decides to ignore it or if he is simply aware that the object is still following him, there is no need to keep including the object in subsequent shots.

The fact that the shot history only keeps the most recent shots means that the event will still be shown to the player from time to time even if he is not facing it. Although the size of the history is limited for performance reasons, the results of this emerging feature seem acceptable as it is quite similar to a filmmaker showing an already seen shot (with possibly a slightly different angle) in order to refresh the audience memory.

- *Automatic Shot Composition*

“The Intruder” does not currently support conversation between characters. However, for demonstration purposes, it is somewhat simulated using the Shot Contribution value: if a character can be talked to, its SC value is greater than zero, thus making it an object of interest for the camera system.

As expected the camera system usually composes the shot in a relevant way. If the avatar moves near a character he can talk to, the system switches from the default following mode to a shot of the two characters (Fig. IV.10a). Additionally, thanks to the event-based system, the transition between two types of scenes is also automatically handled. For example, should a third character joins the conversation, an event is sent and the camera will either dolly in or back to smoothly include the character into the shot or will cut to a new shot showing the thee actors (Fig. IV.10b).

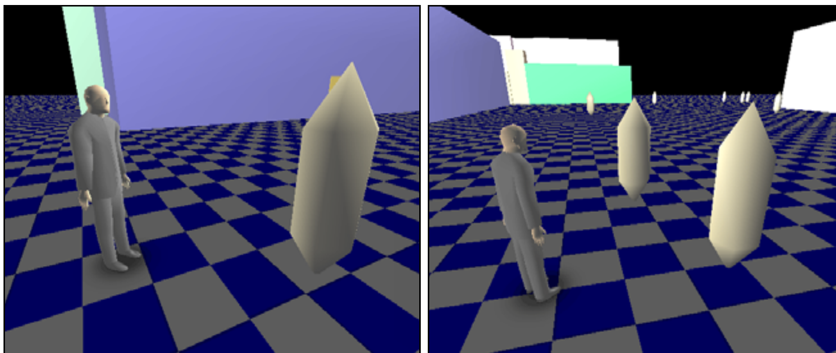

Fig. IV.10a-b. The camera zooms in when the player’s avatar has the possibility to talk to another character; and it automatically composes a new shot if a third character joins the conversation.

If “The Intruder” game engine did support conversation between characters, the rendering of these scenes would be significantly improved without having to modify anything to the current camera system. Indeed each time a character would start talking, the camera system will receive a new event from the story engine and therefore will generate a new shot according to the new situation. Concretely, because the character who is talking has a higher shot contribution value, the camera system is likely to make a long shot of him. Additionally, as the character to whom he is talking is likely to occlude the view, the camera will slightly rotate in order to get the shot right (Fig. IV.11). The result will be an over-the-shoulder shot of the character. And the camera system will do the same each time a character will start speaking thus automatically generating a simple conversation scene with alternate over-the-shoulder shots. This partly demonstrates that the interaction of generic rules (here the “Character Occlusion” and the “New Event” rule), which are not associated with any specific type of scene, can automatically recreate the behaviour of some of the “idioms”.

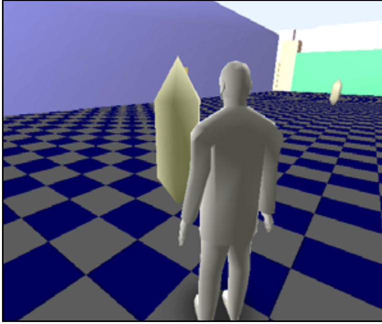

Fig. IV.11. (not implemented) As it is the system can potentially capture a conversation between two characters with automatically generated over-the-shoulder shots.

- *Camera Collision Detection*

In order not to have the camera going through the walls, a simple collision detection solution has been implemented. Once the camera parameters have been set up using the knowledge base, the system checks if the camera is going to collide with an object or a wall. If the camera is below a given distance from any object, it is simply moved away from it by the same distance (Fig. IV.12a-b).

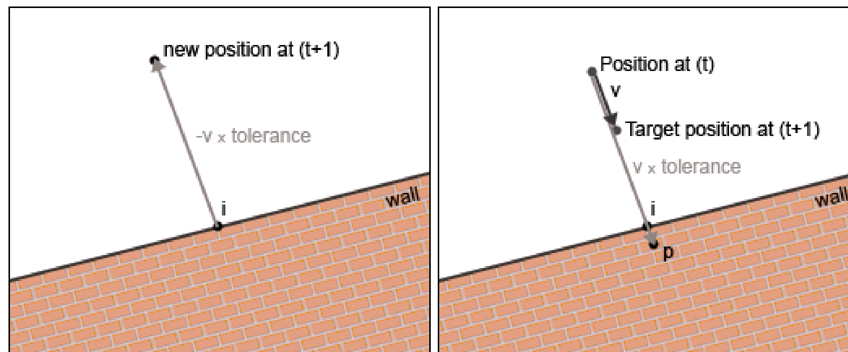

Fig. IV.12a-b. camera collision detection.

- *Handling of Complex Scenes*

In complex scenes involving many objects and especially when a lot of occlusions occur, the camera also tends to jump abruptly from one position to another in an attempt to avoid occlusion. Although it is not a desirable effect, it is an expected behaviour because, as described above, at each attempt to resolve occlusions the constraints are a bit more relaxed which can therefore give less natural shots or camera movements (as the cinematographic rules are less strictly followed).

However in some situations when the camera is only temporarily occluded it would be preferable to keep the current shot as it is. For example when a character quickly passes in front of an important object, there is no need for the camera to suddenly move to show everything. It may be possible to solve this problem relatively easily by using the direction of the objects as a clue to guess their next position and by applying the rules to it. Here the system will generate a temporary shot of the objects with the moving character as it should be in the next few seconds. Then the current shot will be changed only if the camera is still occluded in this forecast shot.

- *System Latency and Minimum Shot Duration*

The system also tends to be "stuck" for too long on some scenes, which means that when the character goes away from it, the camera exaggeratedly dollies in order to keep everything in view (Fig. IV.13). However as written above, this is often useless: if the player decides to go away from a scene (such as a conversation) he is probably not interested in it anymore, which means that the system could already start looking for a new interesting scene or switch to a default tracking shot of the character. This problem is due to the combination of the minimum duration shot and the shot contribution decay rate. Modifying these variables may partly solve the problem but also create new ones. Indeed if the minimum duration is too low, the system may cut and move the camera too often without letting the player grasp each individual shot. In the same way, if the shot contribution decreases too quickly, some important objects may be subsequently ignored by the system. Although not perfect, the current balance between the two parameters seems relatively acceptable in most situations.

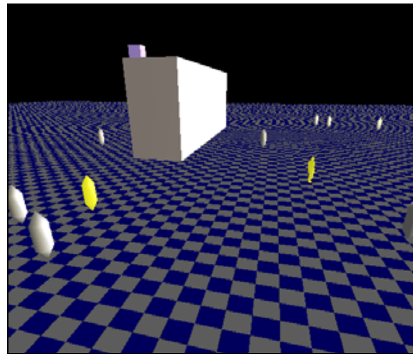

Fig. IV.13. The camera may dolly back too much in an attempt to keep everything in view.

Again due to the minimum shot duration requirements, the system occasionally takes too long before taking into consideration a new event. The result is that the player character may have time to pass near an important object and go away without giving the system time to generate a shot of it. For the same reasons mentioned above this problem is relatively tricky to fix as the minimum shot duration has to be kept reasonably high.

A solution could be to give the possibility to some events to ignore the minimum shot duration requirement and to force the creation of a new shot. But then again this could give strange results where a shot is shown for a few milliseconds before being replaced by the requested one. An alternative solution would be again to try to forecast this kind of situation. By considering the character direction and his speed, we can make an assumption on the important events that are likely to happen in the next few seconds. The length of the shot preceding the possible events could then be adjusted accordingly.

• *Example scenario*

Below is an example scenario meant to illustrate the behaviour of the camera system. It is based on the current demo.

- *Establishing Shot*

The game starts. The system checks if the current scene triggers some of the rules. Each rule can add a shot (with a priority) to the shot list. Once all the rules have been checked, the system selects the shot that has the highest priority.

The first rule is the Establishing Shot rule: it checks if the "important" objects of the scene (those with a SC value greater than zero) have been on screen previously. If none of them has been, it considers that a new scene is starting and queues an establishing shot with a high priority (0.9). In our example, this rule is triggered because, as the game is starting, none of the objects have been seen before.

Additionally, the Default Shot rule is always triggered in case none of the other rules apply to the current situation. It has a low priority (0.0).

Therefore at this stage, there are two potential shots in the shot list, the default shot and the establishing shot. The latter, which has the highest priority, is selected and sent to the constraint solver. Its task will be to find a shot where all the "important" object in the scene are visible.

The resulting shot shows all the objects in the scene.

- *Default shot*

The establishing shot will last for at least one second and for up to ten seconds. After the first second is elapsed, if a new event happens in the scene, the system might end the establishing shot to show the new event. Otherwise it will last for the full ten seconds before switching to the default shot, which shows the character from behind.

- *New event shot*

Later in the demo, the player's character (PC) must meet a non-player character (NPC) to progress in the game. To attract the player's attention on that character, the camera system switches from the default shot to a shot showing the PC and the NPC. At the game editor level, this is done by raising the SC value of the character when the player's character is close to it. The camera system detects the change (through the "New Event" rule of the knowledge base) and therefore triggers a shot that includes the new character (Figure IV.14-abc).

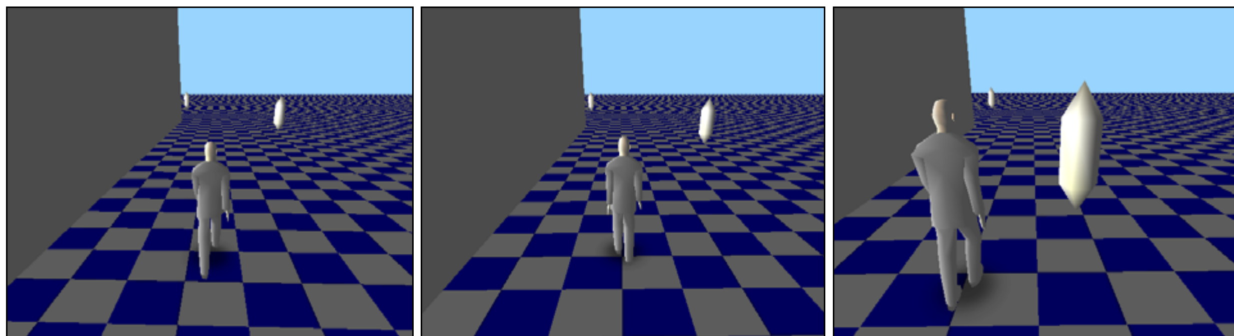

Fig. IV.14a. The NPC waits for the PC to be close enough. Its SC is 0.0.
Fig. IV.14b. The PC is now close enough. The NPC raise its Shot Contribution value to 1.0
Fig. IV.14c. The camera system, through the "New Event" rule, detects the change and switches to a shot of the two characters.

In most videogames, the game designer usually attracts the player's attention by putting the important objects or character in a visible spot (i.e. on the middle of the path, on the player's way). Our system

allows alternatives to this. We can put the NPC on the side or even mixed up in a crowd and still manage to attract the player's attention through camerawork.

- ***Further work***

Although the camera system is already functional, it can still be improved in several different ways. Some user testing could also be done in order to evaluate the system.

- *Predicting the next shot*

Currently, the camera system computes each shot based on the current and previous states of the virtual world. With some extra work it should also be possible to predict what the world state will be in the next one or two seconds. For example, by using the velocity of an object and its direction, it is possible to know where it will be in the next few seconds. Therefore it should be possible to generate future shots based on these predictions.

This could lead to interesting shots being generated automatically. For example, if the character is outside a house at "t1" and the camera system predicts that he will be inside the house at "t2", it would be possible to already start displaying a shot with the camera inside the room showing the character outside. Even if the player decides not to go inside, it would still be an interesting shot as it would give some variations to the default tracking shot.

One another area that would be improved by this kind of predictive camera system is that the shot transitions should be smoother and the shot durations more balanced. For example, the camera could be moved half way between its current position and its next predicted position, so as to smooth the camera movement.

- *Dealing with uneven surfaces*

With a few adjustments, the current camera system should be able to deal with uneven surfaces (at the moment all the characters are on the same horizontal plane). For instance, we could have a conversation between a character on a balcony and another one downstairs, and the camera would still capture the scene. This would demonstrate that the generic rules can adapt to any kind of scene without having to tailor them to a specific scene.

- *User testing*

Finally, some user testing could be done to evaluate the system. Each test could be divided into two parts. First we simply ask the user to play through a game demo without telling him/her what it is about. The camera system will be doing things that are not usual in most games. If the user is able to play through the game without being distracted by the camerawork, it would already mean that at some levels the camera system was successful. It would prove that it is possible to have a more unconventional cinematography without compromising on the gameplay.

A second test could then be performed, this time informing the user that the demo is about testing a camera system and ask him/her to pay attention to the camerawork. This second test could tell us if the user felt he could not see what he needed to see, or if he would have preferred certain shots to be different.

A final test could be done to establish the fairness of the camera system based on the criteria described in this thesis. According to the definition, the user would have to play through the game and count the number of times he/she loses the game because the camera moved at the wrong moment or because one important threat was occluded by another object. Dividing the total of these two types of occurrence by the total number of times the player lost the game and subtracting it from 1.0 would give a clear fairness score that would make it easy to compare one system to another. The higher this score, the better the system in terms of fairness.

V. Conclusion

Through this thesis, various approaches to automated cinematography have been reviewed. The idiom-based approach provides a way to film efficiently certain types of scenes such as a conversation between characters. The results usually adhere to cinematic conventions and allow the user to follow the scene easily. However it has been noted that the fact that idioms can only capture specific types of scenes makes it difficult to adapt the system to complex virtual environments. For example a scene showing a character *randomly* walking in a crowd cannot be described using this system. The alternatives are to create a database large enough to comprise any possible scene or to devise a system that is not dependent on any particular type of scene.

Different games have also been reviewed and it has been outlined that the two main types of camera systems – the predefined views and the free camera – seem to have characteristics that are mutually exclusive. The predefined views allow the game designer to set up the mood of the game and to select the shots in order to create a specific atmosphere; however these objectives occasionally conflict with gameplay. A view that is good in terms of cinematography will not necessarily give the player the best angle to play. The free camera system solves this problem by giving a partial control of the camera to the player, so that he can adjust the view if necessary. However it has been noted that this type of system currently cannot have the emotional expressivity of the predefined views.

I therefore proposed a camera system that attempts to deal with this conflict. One of the main characteristics of this system is that it is driven by the narrative in order to generate shots that are relevant to the plot and/or to the character emotions. To that end, the system has been linked to the story engine, which notifies when something new happen in the scene. The camera system is then able to film the scenes in a more relevant way by selecting what should be in view and what could be ignored.

I have also looked for a more flexible approach than the idiom-based one. My system indeed makes use of rules that are not associated with any specific type of scene. In order to keep them as general as possible, they are only expressed in terms of Objects, Events and Player character. Occlusion detection and resolution is also implemented so that the camera system is able to film any scene.

The main benefit of this system is that it demonstrates that it is possible for a camera system to reconcile the dramatic potential of cinematography with the demand of an interactive experience. The proposed camera system often creates shots that would be unusual in most video games: the camera is not always right behind the character, it is sometime moving around him in order to capture other elements of the scene. For example, when a non-playable character is nearby the main character, the camera adjusts the shot composition in order to include the new character. This is done smoothly, in such a way that the gameplay is not interrupted by the camera movement.

Another benefit is that the system demonstrates that it is possible to create a “smarter” camera that is more aware of its surrounding. Indeed, as the camera is linked to the story engine, it is able to create shots that are relevant to the story. For example, in a virtual city, the camera system will focus on the doors that the player can open and ignore the doors that are part of the décor (with which the player cannot interact or open). This can offer the game designer a new way to highlight objects in the scene. In most games, this is done by putting the object on the player’s path or on a visible spot. This system however allows more flexibility: the designer can put the object anywhere and still manage to attract the player’s attention through camerawork.

The benefits of this system have been outlined in the last part of this thesis. The system allows better anticipation: as the camera rotates towards the interesting parts of the scene, the player can know in advance where he can go. This feature may not be particularly useful in an adventure game as the player may enjoy exploring the virtual environment without this kind of help. However it can be more relevant in a fast paced game where the player has to make quick decisions. For example, in a game such as *Super Mario Sunshine*¸ if the player can have an optimal view of Mario's surrounding after each jump, he will no longer have to control the camera to look for the closest platform.

The fact that the system is linked to the story engine also allows it to quickly react when something new happen in the scene. For example, if a character moves towards the player character or wants to get his attention, the system automatically creates a new shot including the involved characters. The system also attempts to guess what the player is not interested in by using the character's movement and orientation as a clue. For example it has been assumed that if the character turns his back to a scene the player has already seen, it is likely that he is not interested in it anymore. This allows the system to ignore some possible shots to the benefit of other, possibly better, ones.

As it is, my camera system could be adapted to a slow-paced game. For example, it could be used for an adventure game such as a 3D version of *Monkey Island*. However, the system being flexible, it could also be adapted to fast-paced game by modifying the rules. For example in a fighting game, it would be possible to write a single rule that tells the system to focus mainly on the enemies. This will ensure that the fighting scenes are never confusing and are always given the highest priority when generating a shot solution. Then other rules, with lower priorities, could be used to have a camera behaviour similar to the existing one – with a camera showing the avatar's surrounding and the avatar himself – but only when no enemies are nearby.

In the last part of this thesis, it has been noted that one of the features that would enhance the rendering of the scene would be the ability of the system to anticipate the scene in the next few seconds. The current movements of the characters and their speed could allow forecasting the scene relatively precisely in the next few seconds. It would allow smoother transitions between shots by setting the characteristics of the current shots based on those of the forecast one. It would also avoid unnecessary cuts or camera movements when the current shot is similar to the forecast one. The system could also be extended to make it deal with uneven surfaces.

VI. Bibliography

1. Literature

Amerson, Daniel; Kime, Shaun, 2000, *Real-time Cinematic Camera Control for Interactive Narratives*, American Association for Artificial Intelligence, California

Armes R., 1994, *Action and Image: dramatic structure in cinema*, Manchester University Press, Manchester, UK

Bares, W. H.; Thainimit, S.; McDermott S., 2000, *A model for constraint-based camera planning*, Papers from the 2000 AAAI Spring Symposium, Stanford, p84-91

Bares, William H.; Lester James C., 1997, *Cinematographic User Models for Automated Realtime Camera Control in Dynamic 3D Environments*, Proceedings of the sixth International Conference on User Modeling, Sardinia, Italy

Bourne Owen, Satta Abdul, 2005, *Evolving Behaviours for a RealTime Autonomous Camera*, Australasian Conference on Interactive Entertainment, Sydney

Boyd Davis, Stephen; Jones, Huw, 2002, *Screen Space: Depiction and the Space of Interactive Media*, in J.A. Jorge, N.M. Correia, H. Jones and M.B. Kannegai (eds.), Multimedia 2001, Springer, Vienna, p165-176

Boyd Davis, Stephen, 2002, *Media Space: an analysis of spatial practices in planar pictorial media.* PhD Thesis, Middlesex University.

Carroll, Noël, 1996, *Theorizing the Moving Image*, Cambridge University Press

Christie Marc, Normand Jean-Marie, 2005, *A Semantic Space Partitionning Approach to Virtual Camera Control*, Proceedings of the Annual Eurographics Conference, Computer Graphics Forum, Volume 24-3, pp 247-256, Grenoble, France

Cozic, Laurent, 2003, *The Intruder, Expressive Cinematography in Video Games*, working paper, Middlesex University, UK

Drucker, Steven M.; Zelter, David, 1994, *Intelligent Camera Control in a Virtual Environment*, Proceedings of Graphics Interface '94, Alberta, Canada, p190-199

Drucker, S. M., Zeltzer, D., 1995, *CamDroid: A system for implementing intelligent camera control*, Symposium on Interactive 3D Graphics, Monterey, California, p139-144

Friedman, Doron; Feldman, Yishai A., 2004, *Knowledge-Based Cinematography and its Applications*, Proc. 16th European Conf. Artificial Intelligence, Valencia, Spain

Friedman, Doron; Feldman, Yishai A.; Shamir, Ariel; Dagan, Tsvi, 2004, *Automated Creation of Movie Summaries in Interactive Virtual Environments*, Virtual Reality Conference, Chicago

Gessner, Robert, 1968, *The Moving Image*, Cassel & Company Ltd., London

Halper, N.; Helbing, R.; Strothotte, T., 2001, *Computer games: A camera engine for computer games*, Computer Graphics Forum 20

He, Li-wei; Cohen, Michael F.; Salesin, David H., 1996, *The Virtual Cinematographer: A Paradigm for Automatic Real-Time Camera Control and Directing*, Proc. 23rd Int'l. Conf. on Computer Graphics and Interactive Techniques, New York, p217-224

Hornung, Alexander; Lakemeyer, Gerhard; Trogemann, Georg, 2003, *An Autonomous Real-Time Camera Agent for Interactive Narratives and Games*, Proceedings. Lecture Notes in Computer Science 2792 Springer, Germany

Katz, Steven D., 1991, *Film Directing: Shot by Shot. Visualizing from Concept to Screen*, Michael Wiese Productions

Negnevitski, Michael, 2002, *Artificial Intelligence, A Guide to Intelligent Systems*, Personal Education

Poole, S., 2000, *Trigger Happy*, Fourth Estate, London

Reisz, Karel, 1953, *The Technique of Film Editing*, The Focal Press

Tomlinson, Bill; Blumberg, Bruce; Nain, Delphine, 2000, *Expressive Autonomous Cinematography for Interactive Virtual Environments*, Proceedings of the fourth international conference on Autonomous agents, Barcelona, Spain

2. Websites

Chandler, Daniel, 1994, *The 'Grammar' of Television and Film*, The University of Wales Aberystwyth, http://www.aber.ac.uk/media/Documents/short/gramtv.html

Giors, John, 2004, *The Full Spectrum Warrior Camera System*, Gamasutra, http://www.gamasutra.com/gdc2004/features/20040325/giors_02.shtml

Hawkins, Brian, 2003, *Creating an Event-Driven Cinematic Camera (Part 1 & 2)*, Gamasutra, http://www.gamasutra.com/features/20030108/hawkins_01.htm

Perez , Anthony, 2005, N*intendo Revolution Controller Reaction*, Amped IGO, http://igo.ampednews.com/features/142/5/

AIML: Artificial Intelligence Markup Language, The A. L. I. C. E. Artificial Intelligence Foundation, http://www.alicebot.org/aiml.html

Media Glossary, British Film Institute, http://www.bfi.org.uk/education/teaching/disability/further/mediaglossary.html

VII. Games

Alone in the Dark, 1992, PC game, Infogrames, France

Crash Bandicoot, 1996, Playstation game, Naughty Dog, USA

Fear Effect 2, 2001, Playstation game, Kronos Digital Entertainment, USA

Final Fantasy IX, 2001, Playstation game, Square, Japan

Full Spectrum Warrior, 2004, Xbox game, THQ, USA

Half-Life, 1998, PC game, Sierra, USA

Legend of Zelda: Ocarina of Time, The, 1998, Nintendo 64 game, Nintendo, Japan

Legend of Zelda: The Wind Waker, The, 2002, GameCube game, Nintendo, Japan

Morrowind, 2002, PC game, Bethesda Softworks, USA

Neverwinter Nights, 2002, PC game, Bioware Corp., Canada

Populous, 1989, PC game, Bullfrog, UK

Resident Evil 2, 1997, Playstation game, Capcom, Japan

Secret of Monkey Island, The, 1990, PC game, Lucasfilm Games, USA

Sim City, 1989, PC game, Maxis, USA

Super Mario 64, 1996, Nintendo 64 game, Nintendo, Japan

Super Mario Sunshine, 2002, GameCube game, Nintendo, Japan

VIII. Source Code

1. Story Engine

```
global gChar -- reference to the player character
global gCamera -- reference to the camera system

property pScene -- reference to the current scene

-- [NEW] --
-- Creates a new instance of the story engine
-- and initializes its properties
on new me, iScene
  pScene = iScene

  return me
end

-- [SET STORY CONTRIBUTION] --
-- Sets the story contribution value for the given state
-- of the given object
on setStoryContribution me, iObject, iStateID, iSCValue
  iObject.setProperty(iStateID, #storyContribution, iSCValue)
end

-- [UPDATE] --
-- Updates the story engine (on each frame)
on update me
  -- for each object in the scene
  repeat with o in pScene.pList
    activeState = o.getActiveState()

    -- if the object is important enought
    if activeState[#storyContribution] > 0.0 then
      -- if the object is close enought to the character,
      -- informs the camera system of the object proximity
      if activeState.model.worldPosition.distanceTo(gChar.getModel().worldPosition) <= 300 then
        gCamera.newEvent(o, #relevantObjectProximity)
      end if
    end if

  end repeat
end
```

2. Camera System

```
global gGraphicEffects -- graphic library used here for debugging purpose
global gClock

property ancestor --  a reference to the generic entity ancestor

-- PUBLIC
property w -- a reference to the 3D world
property pPC -- a reference to the player's avatar
property pCamera -- the 3D world main camera
property pSCList -- list used to manage the shot contribution value of the important objects
property pCurrentShot -- the current shot properties
property pUniqueID -- a unique ID to identify the shots and other objects
property pInfluenceArea -- if enabled the camera system will only consider the objects inside
this influence area
property pMaxIteration -- Maximum number of iterations allowed to update the camera system (see
#update handler)
property pIterationCount -- Number of iterations since the beginning of the update
property pModelToLookAt -- Invisible model to which the camera is looking at
property pSaveCameraRotation -- Rotation value of the camera at the beginning of the camera
system update
property pCameraProximity -- Specifies how close the camera should be filming the scene
property pCameraCollisionEnabled -- Enables / disables the camera collision detection
```

```
-- CONSTANTS
property MAX_SHOT_DURATION -- passed this duration, the camera system creates a new shot even if
nothing
-- has changed in the scene
property MIN_SHOT_DURATION -- a shot must last at least for this duration (in order to avoid
-- shots of a few milliseconds when many things are simultaneously occurring in the scene)
property SC_DECAY_RATE -- Specifies how fast is going to decrease the shot contribution value of
each object
-- (the decay rate is expressed as the value decremented from the SC per second)
property SHOT_HISTORY_SIZE

-- PRIVATE
property _knowledgeBase -- the camera system knowledge base containing all the cinematographic
rules
property _shotHistory -- history of the recent shots that have been selected
property _shotQueue -- list of possible shots for the current scene
property _startTransform -- used for camera movement - the camera moves and rotates from
_startTransform
property _targetTransform -- to _targetTransform over a given duration

-- [NEW] --
-- Creates a new instance of the camera system and initializes its properties
on new me, i3DMember, iCameraID
  -- Creates the generic entity from which the camera system inherits
  -- allowing it to receive messages from other objects,
  -- and to benefit (if necessary) from the multiple state mechanism.
  ancestor = script("cGenericEntity").new()
  me.setID(1)
  me.setName("Camera System")
  me.setType(#cameraSystem)

  pCameraProximity = 80.0
  pCameraCollisionEnabled = FALSE

  -- Sets the min and max duration for the shots
  -- The minimum shot duration is variable as it depends on the number
  -- of relevant objects included in the current shot (please see below)
  MIN_SHOT_DURATION = 1000
  MAX_SHOT_DURATION = 5000
  SC_DECAY_RATE = 0.8
  SHOT_HISTORY_SIZE = 30

  pUniqueID = 0
  w = i3DMember
  pPC = VOID
  pSCList = []

  -- Sets the camera properties
  pCamera = w.camera(iCameraID)
  pCamera.fieldofView = 70

  -- Initializes the current shot
  pCurrentShot = me.newShot([])
  pCurrentShot.startTime = 0

  -- Initializes the influence area (a circle)
  pInfluenceArea = [#center:VOID, #radius:0]

  -- Initializes the knowledge base, the shot history and the shot queue
  _knowledgeBase = [#shotRules:[], #objectRules:[]]
  _shotHistory = []
  _shotQueue = []

  return me
end

-- [TRACK CHARACTER] --
-- Tells the camera to track the given character
on trackCharacter me, iCharacter
  pPC = iCharacter
end

-- [SET INFLUENCE AREA] --
-- Sets the influence area
-- It's a circle centered on the given character
-- with a radius iRadius
on setInfluenceArea me, iCharacter, iRadius
  pInfluenceArea.center = iCharacter
  pInfluenceArea.radius = iRadius
end
```

```
-- [MAKE SCRIPT] --
-- Creates a new lingo script
-- * iScriptName: Name of the new script
-- * iScriptText: Text of the script in Lingo
on makeScript me, iScriptName, iScriptText
  ruleCastName = "Camera System"

  cm = new(#script, castLib(ruleCastName))
  cm.name = iScriptName
  cm.scriptType = #parent
  cm.preload()
  cm.scriptText = iScriptText

  return cm
end

-- [IMPORT RULE SET] --
-- Import a set of rules into the knowledge base
-- * iFileName: XML containing the rules
on importRuleSet me, iFileName

  -- Deletes any existing rule scripts (for authoring only)
  ruleCastName = "Camera System"

  i = 1
  m = member(i, ruleCastName)
  repeat while m.name <> EMPTY
    m.erase()
    i = i + 1
    m = member(i, ruleCastName)
  end repeat

  -- Reads the XML file
  xmlFile = xtra("fileIO").new()
  xmlFile.openFile(iFileName, 1)
  stringToParse = xmlFile.readFile()

  if voidP(stringToParse) then
    return FALSE
  end if

  xmlFile.closeFile()
  xmlFile = VOID

  -- Parses the XML data
  xmlObject = xtra("XMLParser").new()
  xmlObject.parseString(stringToParse)
  xmlList = xmlObject.makeList()

  xmlList = xmlList["ROOT OF XML DOCUMENT"]["XML"]

  -- For each tag in the XML file
  repeat with listIndex = 1 to xmlList.count

    propName = xmlList.getPropAt(listIndex)

    -- <INITIALIZATION>
    -- Section of the XML file related to the
    -- initialization of the camera system
    if propName = "INITIALIZATION" then
      -- [Read the script]
      initScriptText = xmlList[propName]["!CHARDATA"]

      s = ""

      s = s & "property cs" & RETURN

      s = s & "on new me, iCameraSystemInstance" & RETURN
      s = s & "  cs = iCameraSystemInstance" & RETURN
      s = s & initScriptText & RETURN
      s = s & "  return me" & RETURN
      s = s & "end" & RETURN

      -- [Makes the script]
      scriptName = "cs" & me.getUniqueID() && "Initialization"
      me.makeScript(scriptName, s)

      -- [Run the script]
      initScript = script(scriptName).new(me)
      initScript = VOID
    end if
```

```
    -- <RULE>
    -- Section containing the definition of
    -- a cinematographic rule
    if propName = "RULE" then
      ruleData = xmlList[listIndex]

      -- [Reads the rule properties]
      repeat with attributeIndex = 1 to ruleData["!ATTRIBUTES"].count

        attributeName = symbol(ruleData["!ATTRIBUTES"].getPropAt(attributeIndex))
        attributeValue = ruleData["!ATTRIBUTES"][attributeIndex]

        case attributeName of

          #name: ruleName = attributeValue -- Gets the rule name
          #appliesTo: ruleAppliesTo = attributeValue -- Checks whether the rule applies to
objects or to camera shots

        end case

      end repeat

      if voidP(ruleName) then ruleName = "Unnamed"
      -- Checks if the rule applies to something valid
      if voidP(ruleAppliesTo) then
        alert("Error: Unspecified type for rule <" & ruleName & ">")
        halt()
      end if
      -- Reads the condition script
      ruleConditionScript = ruleData["CONDITION"]["!CHARDATA"]
      -- Read the action script
      ruleActionScript = ruleData["ACTION"]["!CHARDATA"]

      -- [Prepares the rule script]
      s = ""

      s = s & "property cs" & RETURN
      s = s & "property name" & RETURN
      s = s & "property appliesTo" & RETURN

      s = s & "on new me, iCameraSystemInstance" & RETURN
      s = s & "  cs = iCameraSystemInstance" & RETURN
      s = s & "  name =" && QUOTE & ruleName & QUOTE & RETURN
      s = s & "  appliesTo = #" & ruleAppliesTo & RETURN
      s = s & "  return me" & RETURN
      s = s & "end" & RETURN

      s = s & "on checkCondition me" & RETURN
      s = s & ruleConditionScript & RETURN
      s = s & "end" & RETURN

      s = s & "on checkRule me" & RETURN
      s = s & "  if me.checkCondition() then" & RETURN
      s = s & ruleActionScript & RETURN
      s = s & "  return TRUE" & RETURN
      s = s & "  end if" & RETURN
      s = s & "return FALSE" & RETURN
      s = s & "end" & RETURN

      -- [Makes the script]
      scriptName = "cs" & me.getUniqueID() && ruleName
      me.makeScript(scriptName, s)

      -- [Compiles the script and adds it to the knowledge base]
      newRule = script(scriptName).new(me)
      case newRule.appliesTo of
        #objects: _knowledgeBase.objectRules.add(newRule)
        #shots: _knowledgeBase.shotRules.add(newRule)
      end case
    end if
  end repeat

  return TRUE
end

-- [NEW SHOT] --
-- Creates a new shot description (The shot won't necessarily be selected as the current shot)
-- * iObjects: the objects that should be included in the view (if possible)
-- * iPCVisible: tells whether the player's avatar should be visible or not
on newShot me, iObjects, iPriority, iPCVisible
```

```
  -- the player character is always visible unless specified otherwise
  if voidP(iPCVisible) then iPCVisible = TRUE
  -- the priority is optional and is set to 0.5 by default
  if voidP(iPriority) then iPriority = 0.5

  -- creates the new shot
  s = [:]
  s.addProp(#ID, me.getUniqueID()) -- shot unique ID
  s.addProp(#objects, iObjects) -- objects that should be included into the view
  s.addProp(#startTime, the milliseconds) -- the time at which the shot started
  s.addProp(#canChange, TRUE) -- a flag to indicates whether the shot can be changed or not
  s.addProp(#PCVisible, iPCVisible)
  s.addProp(#transition, VOID) -- type of transition between the current shot and this shot
  s.addProp(#shotSize, []) -- (test) the size of each objects on screen
  s.addProp(#priority, iPriority)
  s.addProp(#cameraTransform, VOID)
  s.addProp(#lookAtTransform, VOID)

  repeat with obj in s.objects
    s.shotSize.add(#long)
  end repeat

  return s
end

-- [GET CAMERA] --
-- Returns a reference to the camera
on getCamera() me
  return pCamera
end

-- [NEW EVENT] --
-- Informs the camera system that something new has happened in the current scene
-- * iObject: the object involved in the event
-- * iEventName: the name of the event
-- * iParameters: the event parameters (property list)
on newEvent me, iObject, iEventName, iParameters
  -- gets the active state of the object
  activeState = iObject.getActiveState()
  if not voidP(activeState) then
    -- list of the events that should be ignored by the camera system
    ignoreEvents = [#sceneLoaded]

    if not ignoreEvents.getOne(iEventName) then

      case iEventName of

        #storyContributionUpdate:

          -- if the story contribution has been changed,
          -- changes the shot contribution accordingly
          shotContribution = iParameters.newValue

          -- the camera system will monitor the object if
          -- its shot contribution is greater than 0
          if shotContribution > 0.0 then
            me.monitorObject(iObject, activeState, shotContribution)
          end if

        #relevantObjectProximity:

          -- for now if there's an important object near the character
          -- the camera system monitors it and assigns it a SC of 1.0
          -- Additional checks could be added to assign a more accurate
          -- SC value and to decide whether or not it's actually
          -- useful to monitor the object.
          me.monitorObject(iObject, activeState, 1.0)

      end case

    end if

  end if

end

-- [MONITOR OBJECT]
-- Tells the camera system to monitor the given object
-- effectively adding it to the list of important objects in the current scene
on monitorObject me, iObject, iState, iShotContribution
  addToList = TRUE
```

```
  -- Updates the SC value if the object is already monitored,
  -- if not adds it to the list
  repeat with scData in pSCList
    if iObject = scData.object then -- just updates the shot contribution
      scData.shotContribution = iShotContribution
      addToList = FALSE
      exit repeat
    end if
  end repeat

  if addToList then
    pSCList.add([#object:iObject, #state:iState, #shotContribution:iShotContribution])
  end if

end

-- [COMPARE SHOTS --
-- Compares two shots and returns a number which represents the difference
-- between both. The higher the number the greater the difference.
-- At the moment the function computes the number only by comparing the
-- objects included into the two shots. It may be useful to develop the function
-- to make it consider the position and orientation of the camera and objects as well
on compareShots me, iShot1, iShot2
  objList1 = iShot1.objects
  objList2 = iShot2.objects
  objCount1 = objList1.count
  objCount2 = objList2.count
  output = 0

  if objCount1 = 0 and objCount2 > 0 then
    output = objCount2
  end if

  -- Compares the objects of the first shot...
  repeat with shotIndex1 = 1 to objCount1
    obj1 = objList1[shotIndex1]
    objectID1 = obj1.object.getID()
    diff = 1
    -- ...with those of the second one
    repeat with shotIndex2 = 1 to objCount2
      obj2 = objList2[shotIndex2]
      if obj2.object.getID() = objectID1 then
        diff = 0
        exit repeat
      end if
    end repeat
    -- Updates the difference between the two shots
    output = output + diff
  end repeat

  return output
end

-- [GET CHARACTER ANGLE TO SCENE]
-- Returns the angle between the character and
-- the scene associated with iShot
on getCharAngleToScene me, iShot
  if iShot.objects.count > 0 then
    positionToLookAt = vector(0,0,0)
    objectCenter = 0
    nbObjects = iShot.objects.count
    repeat with i = 1 to nbObjects
      positionToLookAt = positiontoLookAt + iShot.objects[i].state.model.worldPosition
    end repeat

    positionToLookAt = positionToLookAt / nbObjects

    v = positionToLookAt - pPC.getModel().worldPosition

    return v.angleBetween(pPC.getDirectionVector())
  else
    return 0
  end if
end

-- [PRINT SHOT] --
-- Returns various properties of the camera as a string
on printShot me, iShot
  s = s & "Objects:" && iShot.objects.count
  s = s & ", Transition:" && iShot.transition
```

```
  return s
end

-- [UPDATE] --
-- Updates the camera system
on update me

  saveCameraTransform = pCamera.transform.duplicate()

  pCameraCollisionEnabled = member("cameraCollisions").hilite

  -- [SHOT CONTRIBUTION UPDATE]
  -- Decreases the shot contribution value of each
  -- object in the current shot so that objects that have been long enough
  -- on screen can be ignored in subsequent shots if necessary
  objectCount = pSCList.count
  if objectCount > 0 then

    repeat with objectIndex = objectCount down to 1
      -- Gets the shot contribution data
      scData = pSCList[objectIndex]

      -- Decrease the SC value of the object
      scData.shotContribution = scData.shotContribution - SC_DECAY_RATE * gClock.get()
      if scData.shotContribution < 0.0 then scData.shotContribution = 0.0

      if scData.shotContribution <= 0.0 then -- remove the object from the list
        pSCList.deleteAt(objectIndex)
      end if

    end repeat

  end if

  -- [CURRENT SHOT UPDATE]
  -- Only updates the system if the current shot has lasted for at least
  -- minimumShotDuration * number of relevant objects in the shot
  newShotCreated = FALSE

  if the milliseconds - pCurrentShot.startTime > MIN_SHOT_DURATION * (pCurrentShot.objects.count
+ 1) then
    -- Adds the shot to the shot history
    -- for further reference
    pCurrentShot.cameraTransform = pCamera.transform.duplicate()

    me.rememberShot(pCurrentShot)

    -- Empties the shot queue
    _shotQueue = []

    -- [OBJECT RULES]
    -- Finds out which objects are important enough to be
    -- included in the next shot using the knowledge base
    -- cf. the XML file "cs_knowledge_base.txt"
    triggeredRules = []
    repeat with objectRule in _knowledgeBase.objectRules
      r = objectRule.checkRule()
      if r then
        triggeredRules.add(objectRule)
        log("Rule" && QUOTE & objectRule.name & QUOTE && "triggered")
      end if
    end repeat

    -- Selects one of the shot in the shot queue using
    -- the priority value
    if _shotQueue.count > 0 then
      newShotIndex = 0
      repeat with i = 1 to _shotQueue.count
        if newShotIndex = 0 then
          newShotIndex = i
        else
          if _shotQueue[i].priority > _shotQueue[newShotIndex] then newShotIndex = i
        end if
      end repeat

      log("->" && triggeredRules[newShotIndex].name && "selected.")

      pCurrentShot = _shotQueue[1]

      newShotCreated = TRUE
    end if
```

```
end if

-- [CALCULATES THE CAMERA PARAMETERS ACCORDING TO THE CURRENT SHOT DATA]
-- Calculates the position to which the camera should point at
-- Currently it is the mean of all the important object positions
-- including the player character

positionToLookAt = pPC.getModel().worldPosition
objectCenter = 0
nbObjects = pCurrentShot.objects.count
repeat with i = 1 to nbObjects
  positionToLookAt = positiontoLookAt + pCurrentShot.objects[i].state.model.worldPosition
  objectCenter = objectCenter+ pCurrentShot.objects[i].state.model.worldPosition
end repeat

if nbObjects > 0 then
  objectCenter = objectCenter / nbObjects
end if

positionToLookAt = positionToLookAt / (1.0 + nbObjects)

-- Creates the cylinder model resource to which the camera should point at
-- or rotate around
n = "cs_positionToLookAt"
if voidP(w.modelResource(n)) then
  mr = w.newModelResource(n, #cylinder)
  mr.height = 1
  mr.resolution = 32
else
  mr = w.modelResource(n)
end if

-- Sets its radius so that the circle includes all the shot objects
-- including the player character
r = 0
repeat with obj in pCurrentShot.objects
  d = obj.state.model.worldPosition.distanceTo(positiontoLookAt)
  if d > r then r = d
end repeat

d = pPC.getModel().worldPosition.distanceTo(positionToLookAt)
if d > r then r = d

if r <= 40 then r = 40

mr.bottomRadius = r
mr.topRadius = r

-- Creates the model
if voidP(w.model(n)) then
  m = w.newModel(n, mr)
  m.visibility = #front
  m.rotate(90, 0, 0)
  m.worldPosition.z = 1

  m.shaderList.blend = 50
else
  m = w.model(n)
end if

pModelToLookAt = m

if nbObjects > 0 then
  pModelToLookAt.pointAt(objectCenter, vector(0,0,1))
end if

m.worldPosition = positionToLookAt.duplicate()

-- Sets the camera position and rotation
saveTransform = pCamera.transform.duplicate()

if pCurrentShot.objects.count = 0 then
  -- if there are no important objects around the character
  -- the camera simply tracks the character from behind

  pCamera.transform = pPC.getModel().transform.duplicate()
  r = pCamera.transform.rotation
  pCamera.transform.rotation = vector(r.x,20,r.z)
  pCamera.rotate(90, 0, 0)
  pCamera.rotate(0, -90, 0)
```

```
    pCamera.worldPosition = positionToLookAt.duplicate()
    pCamera.translate(0,0,100)
    pCamera.translate(0, 50, 0)
  else
    -- if there are important objects to include in the
    -- current shot:

    if member("CB_cameraLock").hilite then
      pCamera.transform = pModelToLookAt.transform.duplicate()
    else
      pCamera.worldPosition = positionToLookAt.duplicate()
    end if

    r = pCamera.transform.rotation
  end if

  -- Determines how far must be the camera to include all
  -- the objects and characters
  d = me.modelSizeToCameraDistance(pCamera, m, pCameraProximity)

  -- and moves the camera by this amount
  pCamera.translate(0, 0, d)

  -- Saves the initial camera transform which is used later
  -- to smoothly move the camera from its current position
  -- to the new one (when the transition type is #move)
  _startTransform = saveTransform

  m.removeFromWorld()

  -- [SHOT RULES]
  -- Sets the maximum number of iterations
  pMaxIteration = 18
  pIterationCount = 0

  allVisible = FALSE
  iterationCount = 0
  maxIteration = 18
  saveCameraRotation = pCamera.transform.rotation.duplicate()
  pSaveCameraRotation = saveCameraRotation

  -- Checks if the current shot is valid according to the
  -- knowledge base.
  -- Rules may adjust the camera parameters in order to find a better position
  -- if necessary. Objects can also be removed from the shot if the rules
  -- can't be satisfied.
  -- In general, in order not to slow down the game too much, the more iterations
  -- has been made, the less tolerant are the rules, which mean that more objects are
  -- removed and the position of the camera may be significantly changed. Thus the
  -- rules are more likely to be all resolved in the following iteration.

  -- If no solution can be found after maxIteration, the camera system will use the
  -- last generated solution. It should be an unlikely situation but it can happen for
  -- example in narrow corridors where it's difficult to get all the objects into view.
  -- However movements of the objects and characters are likely to change this situation
  -- on the next update.

  -- The number of allowed iteration also determines the resolution of the system.
  -- If the number is high enough (here 18 seems to give acceptable results), the transition
  -- between two shots will be visually smoother. On the other hand, with a too low value, the
  -- system lacks subtlety when looking for a solution (in some case this is due to the fact that
many
  -- important objects are removed from the shots from the first iteration). The transitions
  -- are more abrupt and the camera also tends to shake more frequently.

  allResolved = FALSE
  repeat while not allResolved

    allResolved = TRUE
    -- Checks each rule
    repeat with shotRule in _knowledgeBase.shotRules

      if not shotRule.checkRule() then
        log("Checking" && QUOTE & shotRule.name & QUOTE & "...")
        allResolved = FALSE
      end if
      pCurrentShot.cameraTransform = pCamera.transform.duplicate()

    end repeat

    pIterationCount = pIterationCount + 1
```

```
    if pIterationCount > pMaxIteration then
      allResolved = TRUE
    end if

  end repeat

  -- Finds out which type of transition should be used between the previous shot
  -- and the new shot
  _targetTransform = pCamera.transform.duplicate()

  if voidP(pCurrentShot.transition) then
    shotTransition = me.whichTransition(_targetTransform, saveTransform)
  end if

  pCamera.transform = _startTransform.duplicate()

  --   If it's a cut, move the camera to the new position
  if shotTransition = #cut then
    newTransform = _targetTransform.duplicate()
    log("Transition: #cut")
  else -- if it's a camera movement, interpolate its transform
    -- from the previous transform to the new transform
    newTransform = _startTransform.interpolate(_targetTransform, 10)
  end if

  pCurrentShot.transition = shotTransition

  pCamera.transform = newTransform

  -- For demonstration purpose, highlights the important objects in the current shot
  if member("objectHighlight").hilite then
    repeat with objectInfo in pCurrentShot.objects
      gGraphicEffects.apply(#glow, [#model:objectInfo.state.model, #restart:FALSE,
#duration:1000])
    end repeat
    gGraphicEffects.apply(#glow, [#model:pPC.getModel(), #restart:FALSE, #duration:1000])
  end if

  -- Solves the camera collisions
  if pCameraCollisionEnabled then
    newCameraTransform = pCamera.transform

    dv = newCameraTransform.position - saveCameraTransform.position
    dv.normalize()

    -- Finds out if the camera is at less than 20 units from a wall
    mr = w.modelsUnderRay(pCamera.worldPosition, dv, 1, #detailed)

    if mr.count > 0 then
      mr = mr[1]

      -- If so moves the camera back to a valid position
      if mr.distance <= 20 then
        pCamera.worldPosition = mr.isectPosition + mr.isectNormal * 20
      end if

    end if
  end if

end

-- [GET ITERATION COUNT]
-- Returns the number of iterations in the current camera system update
on getIterationCount me
  return pIterationCount
end

-- [GET MAX ITERATION]
on getMaxIteration me
  return pMaxIteration
end

-- [WHICH TRANSITION]
-- Compares two camera transforms and determines if
-- the system should cut or move the camera between
-- the two shots
on whichTransition me, t1, t2

  maxDistance = 200
  maxRotation = 180
```

```
  if (t1.position.distanceTo(t2.position) < maxDistance) or (t2.rotation - t1.rotation.z >
maxRotation) then
    return #move
  end if

  return #cut

end

-- [MODEL SIZE TO CAMERA DISTANCE]
-- Determines how far the camera should be from the given model
-- for it to take iPercentage% of the screen
on modelSizeToCameraDistance me, iCamera, iModel, iPercentage
  d = iModel.boundingSphere[2] / (sin(degreeToRad(iCamera.fieldOfView / 2.0)))
  d = d / (float(iPercentage) / 100.0)

  return d
end

-- [REMEMBER SHOT]
-- Saves the shot in the shotHistory list
on rememberShot me, iShot
  shotDuplicate = iShot.duplicate()
  _shotHistory.add(shotDuplicate)

  if _shotHistory.count > SHOT_HISTORY_SIZE then _shotHistory.deleteAt(1)
end

-- [PRINT]
-- Returns as a string various information
-- about the system state
on print me
  s = ""

  repeat with scData in pSCList
    if s <> "" then s = s & RETURN
    s = s & scData.object.getName() & "," && scData.state.ID & ":" && scData.shotContribution
  end repeat

  s = s & RETURN & RETURN
  s = s & "Shot ID =" && pCurrentShot.ID & RETURN
  s = s & "Shot Duration =" && (the milliseconds - pCurrentShot.startTime) & RETURN
  sl = ""
  repeat with obj in pCurrentShot.objects
    if sl <> "" then sl = sl & ", "
    sl = sl & obj.object.getName()
  end repeat
  s = s & "Objects =" && sl

  return s
end

-- [GET RELEVANT OBJECTS]
-- Returns the list of relevant objects in the current scene
on getRelevantObjects me
  return pSCList.duplicate()
end

-- [COUNT NUMBER OF TIME ON SCREEN]
-- Counts how many times the object has been part of a shot.
-- It can be a useful indicator to solve the visual constraints: if an object
-- has already been seen many times on screen it may be possible
-- to remove it from the current shot in order to relax the constraints.
on countNumberOfTimesOnScreen me, iObject
  c = 0

  if _shotHistory.count <= 0 then

  else
    repeat with shot in _shotHistory
      shotObjects = shot.objects
      repeat with obj in shotObjects
        if obj.object.getID() = iObject.object.getID() then
          c = c + 1
        end if
      end repeat
    end repeat
  end if

  return c
```

```
end

-- [QUEUE SHOT]
-- Adds a shot to the shot queue
on queueShot me, iShot
  _shotQueue.add(iShot)
end
```

3. Knowledge Base Sample

Below is the current knowledge base of the camera system.

```
<XML version="1.0">

<INITIALIZATION>

    -- In this section goes everything that needs to be initialized
    -- before the camera system starts

    -- "cs" is a reference to the camera system

    -- Sets the minimum shot duration
    cs.MIN_SHOT_DURATION = 1000 -- one second
    -- Sets the maximum shot duration
    cs.MAX_SHOT_DURATION = 10000 -- ten seconds

</INITIALIZATION>


<RULE name="Establishing Shot" appliesTo="objects">

<CONDITION>

    -- This rule checks the number of time each object in the scene
    -- has been on screen before

    -- If no object has been on screen before, it means that
    -- it's a new scene - therefore an establishing shot
    -- would be suitable.

    -- Gets the list of relevant objects in the current scene
    relevantObjects = cs.getRelevantObjects()

    -- If there's no important object in the scene
    -- or near the character, don't trigger the establishing shot:
    if relevantObjects.count = 0 then return FALSE

    -- otherwise, checks if the scene has just started
    isNewScene = TRUE
    repeat with object in relevantObjects
        if cs.countNumberOfTimesOnScreen(object) > 0 then
            isNewScene = FALSE
            exit repeat
        end if
    end repeat

    return isNewScene

</CONDITION>

<ACTION>

    -- Gets all the relevant objects
    relevantObjects = cs.getRelevantObjects()

    -- composes a new shot with them
    -- and assigns the shot a high priority
    newShot = cs.newShot(relevantObjects, 0.9)

    -- and queue it
    cs.queueShot(newShot)

</ACTION>

</RULE>
```

```
<RULE name="New Event Shot" appliesTo="objects">

<CONDITION>

    -- This rule simply checks if something new has happened
    -- and if so creates a new shot with the objects involved
    -- in the new event.
    relevantObjects = cs.getRelevantObjects()

    return relevantObjects.count > 0

</CONDITION>

<ACTION>

    -- Creates a new shot with the objects involved in
    -- the new event
    newShot = cs.newShot(cs.getRelevantObjects(), 0.8)

    -- If the event has already been seen and if the player's
    -- character is not facing it, the system assumes
    -- that the player is not interested in it anymore
    -- and therefore does not queue the new shot.
    addShot = TRUE
    shotHistory = cs.getShotHistory()

    repeat with oldShot in shotHistory
        -- if the shot has already been generated:
        if cs.compareShots(oldShot, newShot) = 0 then
            -- Don't add the shot if the angle between the new event
            -- and the character is greater than 120
            if greaterThan(cs.getCharAngleToScene(newShot), 120) then
                addShot = FALSE
                exit repeat
            end if
        end if

        if not addShot then
            exit repeat
        end if
    end repeat

    if addShot then
        cs.queueShot(newShot)
    end if

</ACTION>

</RULE>


<RULE name="Default Shot" appliesTo="objects">

<CONDITION>

    -- The purpose of this rule is to provide the camera system
    -- with a default shot in case none of the other rules
    -- apply to the current situation.

    return TRUE

</CONDITION>

<ACTION>

    -- Creates a shot with no objects other than the character
    -- and assigns it a priority of 0 so that the shot will
    -- be selected only if no other shot is available.
    newShot = cs.newShot([], 0.0)

    cs.queueShot(newShot)

</ACTION>

</RULE>


<RULE name="Object Visibility" appliesTo="shots">

<CONDITION>
```

```
    -- This rule checks if all the objects are visible
    -- in the current shot. If not it will modify the shot
    -- in order to find a better solution. At the moment this
    -- is done by progressively removing on each iteration
    -- the less important objects from the shot.

    -- Sets a new global variable which will store the
    -- result of this rule, as it is used by other rules such
    -- as the "180 degree rule".
    setVar(#objectsVisible, FALSE)

    -- Gets the variables needed by the rule
    maxIteration = cs.getMaxIteration()
    iterationCount = cs.getIterationCount()
    currentShot = cs.getCurrentShot()
    objectCount = currentShot.objects.count
    iterationCount = 0
    camera = cs.getCamera()

    allVisible = TRUE

    -- For each object in the current shot (minus those that have been
    -- removed on the previous iterations)
    repeat with objIndex = 1 to objectCount
    --repeat with objIndex = 1 to objectCount - (iterationCount - 1) / maxIteration
        -- Gets the object from the shot
        obj = currentShot.objects[objIndex]

        -- Gets its 3D model
        objectModel = obj.state.model

        -- Casts a ray from the camera in the direction of the object
        -- and gets the list of objects that have been intersected.
        dv = objectModel.worldPosition - camera.worldPosition
        dv.normalize()
        ml = cs.w.modelsUnderRay(camera.worldPosition, dv, 1)

        -- Checks if there's at least one model among the intersected
        -- models and checks that it's the current object model
        if lowerThan(ml.count, 0) then
            allVisible = FALSE
        else
            allVisible = ml[1].name = objectModel.name
        end if

        -- If at any point one of the object is not visible,
        -- exit the rule. On the next iteration, one more object
        -- will be removed from the shot.
        if not allVisible then
            exit repeat
        end if
    end repeat

    return allVisible

</CONDITION>

<ACTION>

    -- Used to tell the other rules, that
    -- all objects are visibles.
    setVar(#objectsVisible, TRUE)

</ACTION>

</RULE>


<RULE name="Character Visibility" appliesTo="shots">

<CONDITION>

    -- This rule checks that the character is visible on
    -- the current shot.
    maxIteration = cs.getMaxIteration()
    iterationCount = cs.getIterationCount()
    currentShot = cs.getCurrentShot()
    objectCount = currentShot.objects.count
    iterationCount = 0
    camera = cs.getCamera()
```

```
    -- Sets a new global variable which will store the
    -- result of this rule for further reference.
    setVar(#characterVisible, FALSE)

    objectModel = cs.pPC.getModel()
    dv = objectModel.worldPosition - camera.worldPosition
    dv.normalize()
    ml = cs.w.modelsUnderRay(camera.worldPosition, dv, 1)

    if lowerThan(ml.count, 0) then
        charVisible = FALSE
    else
        charVisible = ml[1].name = objectModel.name
    end if

    return charVisible

</CONDITION>

<ACTION>

    setVar(#characterVisible, TRUE)

</ACTION>

</RULE>


<RULE name="180 Degree Line" appliesTo="shots">

<CONDITION>

    -- This rule ensures that the 180 Degree Line is not crossed
    -- between two successive shots.
    -- If first checks if everything is visible, and it not it moves the camera
    -- to a different position while respecting the 180 rule.

    -- (NB: It may be better to put the visibility constraints solving
    -- in the [action] section of the Character and Object visibility rules.)

    -- Gets the variables needed by the rule
    modelToLookAt = cs.pModelToLookAt
    iterationCount = cs.getIterationCount()
    maxIteration = cs.getMaxIteration()
    camera = cs.getCamera()
    previousCameraTransform = camera.transform.duplicate()

    -- First, the rules checks if all the objects and the character are visible
    -- if not the camera will rotate for up to 90 degree clockwise around
    -- the barycenter defined by the positions of the objects. If no solution
    -- can be found that way, it will do the same counterclockwise thus preserving
    -- in both cases the constraint integrity.
    if not var(#objectsVisible) or not var(#characterVisible) then
        modelToLookAt.addToWorld()

        if lowerThan(iterationCount, maxIteration / 2) then
            r = -180 / (maxIteration / 2)
        else
            if iterationCount = maxIteration / 2 + 1 then
                camera.transform.rotation = cs.pSaveCameraRotation
            end if
             r = +180 / (maxIteration / 2)
        end if

        camera.rotate(0,r,0, modelToLookAt)

        modelToLookAt.removeFromWorld()

        return FALSE
    else
        return TRUE
    end if

</CONDITION>

<ACTION>

</ACTION>

</RULE>
```

</XML>

IX. Glossary of Cinematographic Terms

The glossary in based on the *British Film Institute Media Glossary* and *The 'Grammar' of Television and Film* by Daniel Chandler (1994). The definitions of Frame, Shot and Scene come from *The Moving Image* by Robert Gessner (1968)

180º Line or **Line of Action**

An imaginary line used to help stage camera positions for shooting action. Typically 'drawn' along the line of sight between two characters in a scene, or following the movement of characters, cars etc.

Depth of Field

The distance between the objects nearest and furthest from the camera that will be in acceptably sharp focus.

Dolly

When the camera is moved towards (dolly in) or away (dolly back) from the subject.

Frame

A single composition within a shot wherein space has been shaped and motion is implied.

Jump cut

Abrupt switch from one scene to another.

Low-angle shot

When the camera is below the character, exaggerating his or her importance.

Over-the-shoulder Shot

A shot framed by the side of the head and shoulders of a character in the extreme foreground, who is looking at the same thing we are - usually another character in a dialogue sequence.

Pan

When the camera pivots on its vertical axis; the shot that results from this. From panorama or panoramic.

Scene

In cinema the scene is composed of shots so arranged as to express a minor dramatic climax or an expository statement.

Shot

Composed of frames, the result of a single camera operation, its length determined by editing.

Subjective View or **Point of View shot**

A shot where we appear to be looking through the character's eyes, from his or her point of view.

Tilt

When the camera pivots on the horizontal axis; the shot that results from this.

Zoom

The change of image size achieved when the focal length of the zoom lens is altered.

X. The Intruder – Expressive Cinematography in Videogames