

Advanced Composition in Virtual Camera Control

Thesis by
Rafid K. Abdullah

In Partial Fulfillment of the Requirements
for the Degree of
Master of Philosophy



School of Computing Science
Newcastle University
Newcastle upon Tyne, UK

2011

To my father and mother.

Thank you for your support and encouragement at every step of my life.

To my brothers, Waleed and Sarmad.

Thank you for showing me the path to computing science.

Acknowledgements

I would like to thank professor Patrick Olivier from Newcastle University and Dr. Marc Christie from the University of Nantes for their supervision. Special thanks to professor Olivier for putting so much effort in reviewing the thesis which helped me present it in the best possible way.

I would like to acknowledge the Integrated Research in Interactive Storytelling (IRIS) project, the funding of which made my study possible. I would also like to thank Ahmed Kharrufa for recommending me to carry out this research. Thanks are also due to my brother, Zaid Abdulla, for his invaluable help in the technical aspects of this research.

I forward my thanks to Phil Heslop, Guy Schofield, Gavin Wood, Paul Dunphy, and Ahmed Kharrufa from Culture Lab for proof reading the thesis. I also forward my thanks to everyone else in Culture Lab for making my study time such a wonderful time.

Finally, I would like to express my gratitude to my family. I would not have been where I am today without their invaluable support since childhood.

Abstract

Rapid increase in the quality of 3D graphics coupled with the evolution of hardware capabilities urges the development of camera control systems that enable the application of aesthetics rules and conventions from photography and cinematography. One of the most important problems in photography and cinematography is that of composition, the precise placement of elements in shot. Researchers already considered this problem, but mainly focused on basic compositional properties like size and framing. This thesis presents a camera control system that considered advanced composition conventions from photography and cinematography. A number of rules from the literature of photography and cinematography were selected for implementation and were rated according to some functions, allowing the use of optimisation to find the best possible camera configuration. To get accurate results, image processing methods were used in the rating functions, rather than geometric approximations like bounding boxes as employed by many camera systems. Finally, to allow the implementation of more composition rules, a language was implemented that enables the user to write rules depending on image processing operators.

Contents

Acknowledgements	iii
Abstract	iv
1 Introduction	1
1.1 Key Problems in Camera Control	2
1.1.1 Complexity (Non-linearity) of Camera Equations	2
1.1.2 Occlusion	3
1.1.3 Object Abstraction	3
1.1.4 Frame Coherency	3
1.1.5 Cinematography	5
1.2 Applications of Camera Control	5
1.2.1 Games	5
1.2.2 Scientific Visualisation	6
1.2.3 Interactive Storytelling	6
1.3 Composition in Camera Control	7
1.3.1 Graphic Controls	8
1.3.2 Photographic Controls	8
1.3.3 Colour Controls	8
1.3.4 Total Controls	9
1.4 Towards Composition in Camera Control	9
2 Literature Review	10
2.1 Classification of Camera Control	10
2.1.1 Interactive Control	10
2.1.1.1 Through-the-Lens Camera Control	11
2.1.2 Automatic Control	12
2.2 Occlusion in Camera Control	13
2.2.1 Occlusion Detection	13
2.2.2 Occlusion Avoidance	14
2.3 Abstraction	15

2.4	Target Objects Limit in Camera Control	16
2.5	Flexibility Problem in Camera Control	16
2.6	Camera Control in Applications	17
2.6.1	Camera Control in Games	17
2.6.2	Camera Control in Cinematography	18
2.7	Composition in Camera Control	19
2.8	Conclusion	22
3	Formulating the Composition Rules	23
3.1	Selecting Rules for Implementation	23
3.2	Rating Rules	25
3.2.1	Geometric Approach vs. Rendering Approach	26
3.2.2	Visibility Issue	26
3.2.3	Silhouette Rendering	26
3.2.4	Rating Functions	28
3.2.5	Shot Processing	31
3.2.6	Rule Modules	32
3.3	Texture Resolution	32
3.4	Depth of Field Effect	33
3.5	Combining Rules Rating	34
3.6	Conclusion: Limitations in Implemented Rules	35
4	Optimisation Framework	36
4.1	Requirements of the Optimisation Method	37
4.2	Particle Swarm Optimisation	38
4.3	Initial Distribution of Particles	39
4.4	Search Spaces	40
4.5	Multi-rendering	40
4.6	Constraining PSO	41
4.7	Tuning PSO Parameters	42
4.8	The Big Picture of the System	43
4.9	Example: Simple Rules	44
4.10	Performance Analysis: 1984 Canteen Scene	48
4.11	Conclusion	49
5	C for Camera Composition	52
5.1	Requirements	52
5.2	Variable Types	54
5.3	Built-in Functions	54

5.3.1	Image Processing Functions	54
5.3.2	General-use Functions	55
5.4	External Communication	56
5.5	Direct Image Access	56
5.6	Standard Requirements	57
5.7	Interpreter	57
5.8	User Rule Module	58
5.9	Practical Example: Rule of Thirds	58
5.10	Conclusion	61
6	Conclusion	62
6.1	Summary	62
6.2	Limitations	63
6.3	Future Work	64
A	Ccc Interpreter	66
A.1	Lexical Analyser	66
A.2	Grammar	67
B	Vertex and Fragment Programs	70
	Bibliography	72

List of Figures

1.1	Illustrating the key problems of camera control.	4
1.2	According to the Rule of Thirds, it is aesthetically better to place the main subject of interest on any one of the four power corners made by two equally spaced horizontal and vertical lines.	7
2.1	Illustrating Blinn's method. Image taken from [Blinn, 1988].	12
2.2	Halper et al. (2001) renders depth information from the camera towards the target, then render a set of objects of preferred locations in the scene with depth test enabled. The result is that occluded areas will have black colours, while non-occluded areas will have white colour with varying brightness according to preference, among which the brightest point is chosen, and depth-buffer inversion is used to get a non-occluded camera position. The image is taken from [Halper et al., 2001]	18
2.3	Different camera placements for a two-character conversation. This image is taken from [He et al., 1996], which is adapted from [Arijon, 1976].	19
2.4	The semantic volumes around two characters in the cinematography system of Lino et al. [Lino et al., 2010], from which this image is taken.	20
3.1	Images illustrating composition rules.	24
3.2	To make image processing easier, the camera system only render ruled objects and use different colour for each object. Furthermore, it brings the pixels resulting from rendering towards the centre of the image such that the horizontal and vertical distances of the pixel to the centre of the image are half their original values. This way the camera system can process pixels which are originally out of view.	27
3.3	The camera system uses a field of view wider than the original field of view so that it can process pixels which are out of view. However, the configuration in this figure shows an example of a case in which the object is large enough to be fully contained in the new field of view. In such cases, the calculated visibility is larger than the actual visibility.	28
3.4	Illustrating the two factors determining the rating of the diagonal dominance rule. .	29

3.5	Illustrating how the prominent line of an object is found using perpendicular linear regression. The line in blue is the prominent line extracted from the object in dark red.	32
3.6	The connection between the different module in the camera system.	33
4.1	Colour maps representing the achieved rating for different values of the PSO parameters for one of the configurations used in the test from section 4.7, where red pixels represent better achieved rating and blue pixels represent worse achieved rating. Each colour map represents a different value of ω , and for each map, the horizontal scale represents increase in cognitive factor from 0.2 to 2.6 in 0.2 steps, while the vertical scale represents increase in social factor on the same range.	43
4.2	Colour maps representing the achieved rating for different values of the PSO parameters for the other of the configurations used in the test from section 4.7, where red pixels represent better achieved rating and blue pixels represent worse achieved rating. Each colour map represents a different value of ω , and for each map, the horizontal scale represents increase in cognitive factor from 0.2 to 2.6 in 0.2 steps, while the vertical scale represents increase in social factor on the same range. . . .	44
4.3	Illustrating the subcomponents of the camera system and the communications among them.	45
4.4	Snapshots of Example 1	47
5.1	Having regular expressions specifying tokens and grammar rules, Lex & Yacc can be used to generate an interpreter that convert source code to byte code.	58
5.2	The complete camera system.	59

List of Tables

3.1	The factors the camera solver depends on to rate the rules.	30
3.2	Symbols and abbreviations used in factors calculation.	30
4.1	Comparison between the normal penalty method and the reset-penalty method. . .	42
4.2	The list of shots used in the rendering of a scene from Nineteen Eighty-Four. For each shot, we repeated the test 10 times and calculated the average rating the system could achieve and the average time spent in the solving process. The numbers in the brackets are the standard deviation of the results of the 10 tests.	50
4.3	The time in seconds spent in the different parts of the solving process for single-rendering.	51
4.4	The time in seconds spent in the different parts of the solving process for multi-rendering.	51
5.1	The operators supported in Ccc	57
5.2	The time in seconds spent in the different parts of the solving process for multi-rendering.	61

Chapter 1

Introduction

Virtual camera control is the process of positioning, orienting, and configuring lens parameters of a virtual camera to achieve certain screen goals. For example, in a film dialogue the camera should be configured in a way that clearly identifies the actors and the environment, and the relations and interaction among them. Another example is visualisation; in a visualisation system, the camera should be configured such that it shows the necessary details of the visualisation geometry.

At the dawn of computer graphics, virtual environments consisted of only a few simple objects and virtual camera control was an easy task consisting of positioning and orienting the camera such that it shows the main object at some specified position on screen. But the rapid progress in computer graphics and animation made it possible to create rich virtual environments, thus opening the door for the implementation of more sophisticated 3D applications. Different applications impose different requirements on the virtual camera controller. For example, in interactive applications, an essential requirement of the virtual camera controller is to run in real-time.

In general, there are two approaches to camera control: procedural and declarative. *Procedural control* means controlling the camera through certain commands that have direct control on the camera parameters. For example, the command *look at* directly changes the orientation of the camera such that an object is at the centre of the screen. On the other hand, *declarative control* is the control of camera parameters such that certain screen goals are achieved. For example, the user might request that an object be placed on a certain frame in the screen. Here the user does not specify the parameters directly, but rather specify the intended goals and the camera system is responsible for controlling the camera parameters such that the goals are satisfied.

In this chapter, the key requirements of today's virtual camera controllers are discussed, followed by a discussion of the most important applications of virtual camera control and the requirements of each. Throughout this thesis (unless otherwise stated) the term *camera control* will be used to refer to *virtual camera control* rather than *physical camera control* as it is usually understood outside the context of computer graphics. Moreover, the terms *camera controller*, *camera control system*, *camera system* are used interchangeably to refer the same thing.

1.1 Key Problems in Camera Control

The process of camera control, although seemingly simple, has proven to be very difficult to implement in practice for a number of reasons. The most important of these reasons are the strong non-linearity of camera equations, occlusion, object abstraction, coherence, and cinematography.

1.1.1 Complexity (Non-linearity) of Camera Equations

The standard method of representing a camera consists of specifying the position of the camera, the direction vector of the camera, and an up vector specifying the rotation of the camera around the direction vector. In addition, the camera model requires an angle specifying the camera field of view. Based on this, a transformation matrix is formed that transforms the coordinates of object vertices from *world coordinates* to *screen coordinates* so that it can be rendered on screen. The transformation matrix is the concatenation of a *translation matrix* that translates the world such that the camera position is at the origin, a *rotation matrix* that rotates the world such that the direction vector coincides with the negative z-axis, and a *projection matrix* that converts from 3D coordinates to 2D screen coordinates. Thus a standard camera is represented by the following equation:

$$\begin{pmatrix} x_s \\ y_s \end{pmatrix} = P(\phi) \cdot R(d_x, d_y, d_z) \cdot T(x_c, y_c, z_c) \cdot \begin{pmatrix} x_w \\ y_w \\ z_w \\ 1 \end{pmatrix} \quad (1.1)$$

where

(x_w, y_w, z_w) : the coordinates of a point in world coordinates

(x_c, y_c, z_c) : the coordinates of the camera in world coordinates

(d_x, d_y, d_z) : the coordinates of the camera direction vector

ϕ : an angle specifying the camera field of view

(x_s, y_s) : the coordinates of the point after projection

This equation gives the screen coordinates of a vertex in world coordinates, given the camera parameters (position, direction, and field of view). However, what is required in camera control is the inverse. That is, it is required to get the camera parameters given the intended position of a vertex on the screen; given (x_s, y_s) , we need to solve equation 1.1 for camera parameters. In addition to being strongly non-linear, making it very difficult to solve, this equation is under-constrained with seven variables but only two variables to solve for, meaning that an infinitely many solutions exist.

To make the problem easier, many researchers [Blinn, 1988, Christie and Marchand, 2000, Gleicher and Witkin, 1992] and game developers placed constraints on some of the camera parameters. However, the significantly greater complexity arises when there are *many* points to be placed

on specific positions on the screen, making it practically impossible to solve the equation analytically.

1.1.2 Occlusion

A challenging problem in camera control is occlusion. Occlusion happens when some objects in the scene are between the camera and objects of interest. For example, in computer games, the main player might be occluded by a column between the camera and the player (figure 1.1a.) Similarly, in scientific visualisation, the important details of the geometry might be covered by other details of the geometry. Depending on whether the objects of interest are partially or totally covered, the occlusion is called *partial* or *total* occlusion.

There are mainly two cases in which occlusion happens. The first case is when the camera system only considers target object(s) while calculating the position of the camera. In this case, any object which happens to lie between the calculated camera position and the target object(s) causes partial or total occlusion. The second case is when the camera system does consider other scene objects, but scene objects are approximated as points or simple geometric shapes, making the computation inaccurate which might produce partial occlusion.

The problem is further complicated in dynamic scenes, in which either the target objects or the camera is (are) moving. In this case it is not enough to consider the occlusion at the current instant, and the timeline of the motion should be considered.

1.1.3 Object Abstraction

The problem of abstraction happens when objects in scene are approximated by simple geometric shapes, resulting in accuracy problems. The reason for using simple geometric shapes is to simplify calculations. For example, in some camera control systems that use vector algebra [Blinn, 1988, Christianson et al., 1996], the objects are represented by points. In such camera systems, an object, for example, determined to be bounded by a certain frame on screen might in fact occupy areas beyond the frame since the camera system only determined the point representing the object to be bounded by the frame, rather than the full extent of the object on screen. For an example of a problem caused by abstraction, see figure 1.1b. Other examples include the use of axis-aligned bounding boxes and bounding spheres to approximate scene objects, both of which overestimate occlusion.

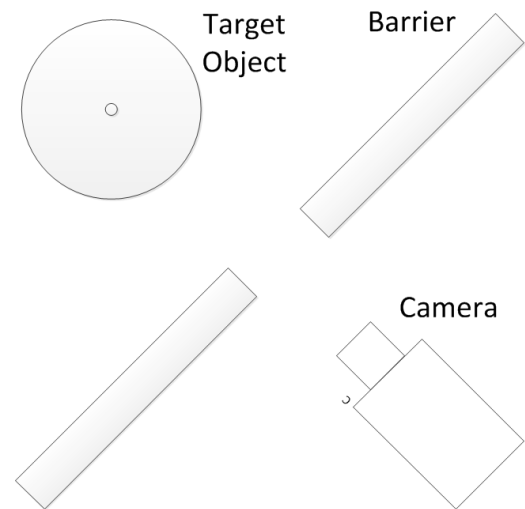
1.1.4 Frame Coherency

Frame coherency is the maintenance of smooth changes in camera position and orientation. As Halper et al. (2001) pointed out, maintaining frame coherency is necessary to avoid disorienting players. For example, a camera system which is purely reactive to changes in scene or shot requirements might produce jumpiness that disorients the player or spectator (figure 1.1c). To

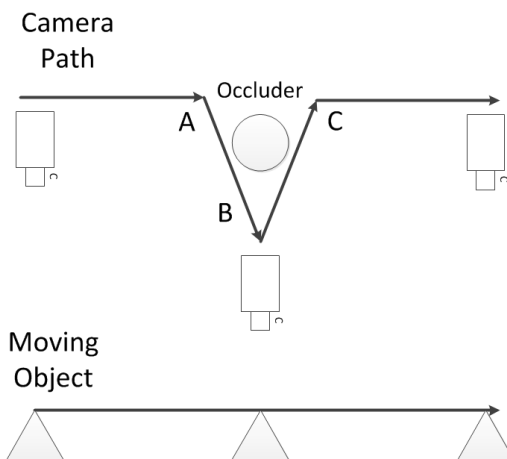


Camera

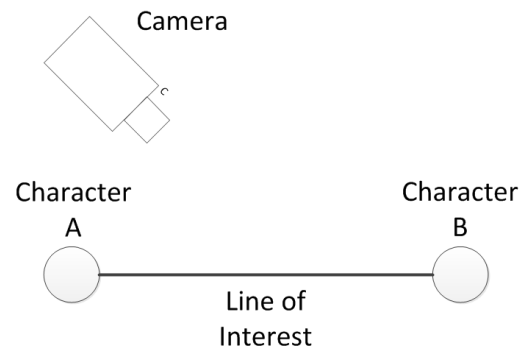
(a) A basic approach to directing the camera to an object is to place the camera on the circle centred around the object with a certain radius, then orient the camera towards the object. But without checking the environment, this might generate occlusion problem if there is an occluder between the camera and the object as shown in this figure.



(b) Treating objects as points might produce unexpected results. For example, in this configuration, the calculation shows that there is no occluder between the camera and the object (treated as a point) when in fact the target is partially occluded by the barrier.



(c) Instant reaction to changes in scene might produce jumpiness in camera movement. In this configuration, there is an occluder in the path of the camera. If the camera system reacts instantly to this occluder, it will jump from position A to position B.



(d) For more expressive shots, cinematographers have developed a set of rules for the camera movement. In this illustration, since the camera started in the side above the line of interest, it should not move to the other side to avoid confusing the user or spectator.

Figure 1.1: Illustrating the key problems of camera control.

maintain frame coherency, the camera system is required to find the position and orientation of the camera based on previous positions and orientations.

1.1.5 Cinematography

For more expressive shots, cinematographers have developed a set of rules for the camera placement and movement in a film. For example, in a conversation between two actors, the camera should not cross the line connecting the two actors, which is called the *line of interest*, otherwise the actors' positions on screen will be switched, confusing spectators [Arijon, 1976] (figure 1.1d). These rules enforce constraints on the camera movements and add more complexity to the process of controlling the camera. For example, to avoid crossing the line of interest half of the search space cannot be utilised.

One of the most important inventions in cinematography is the notion of an *idiom*, a conventional style of shooting a certain action in a film. For example, in a dialogue between two actors, a film maker might start with a long view of both characters to show the environment the actors are in, then gradually move the camera closer to the actors. Such conventions impose extra constraints on the camera placement as it means the camera position should be calculated based on previous positions. In other words, the task of camera control raises to the level of finding *camera paths* satisfying certain rules and screen properties, rather than just camera configurations satisfying certain screen properties.

1.2 Applications of Camera Control

Depending on the application context, different 3D applications impose different requirements on camera controllers. In this section three different important application contexts and the requirements of the camera controllers for each of them are explained.

1.2.1 Games

Although there are many genres of video games, a feature that most of them share in common is high interactivity, making real-time operation an essential requirement in the camera controller. Based on the genre of the game, the camera control may be manual or interactive. For example, in strategy games where players have full control over units in the game, the user is expected to have manual control over the camera. The canonical camera control of such games usually includes panning, zooming, and sometimes view rotation through the mouse or the keyboard.

On the other hand, action games require the camera to be interactive based on the movement of the player. For example, in first-person shooter (FPS) games, the view of the game shows how the player see the world as a way of reflecting the feeling of being in the game. This makes the task of implementing the camera system straightforward since the camera position and orientation can simply be assigned the same values as those of the player's head. However, in third-person shooter (TPS) games the camera is usually behind the player and slightly above him/her, imposing certain requirements on the camera, specifically occlusion avoidance. For in many cases the player

might be in places (e.g. backs-up against a wall) where having the camera behind the player hides the player. Many games in such cases usually change the camera to frontal view, thus hiding the opponents and confusing the player.

In general, unless the camera is strictly bound to the player's position, frame coherency is another essential requirement of camera control in games. For example, in a third-person shooter game, if the player shifts in front of a rock, the camera might move upwards to avoid occlusion. But if this happens instantly, it will produce jumpiness that confuses the player. Finally, with the rapid increase in graphics richness, game developers started adopting rules of cinematography to increase the expressiveness of the gameplay.

1.2.2 Scientific Visualisation

Scientific visualisation is the presentation of data in a visual form, usually as a three-dimensional entity, to make the process of exploring and understanding the data easier. Typically, scientific visualisation is needed when the size of the data to be explored is massive and it is required to get an insight into the big picture of the data. Additionally, the geometry of the data to be visualised is usually complicated.

Due to the usual massive size and complicated geometry of the data, manual control of the camera is difficult and a camera control which enables the user to easily explore the data is thus required. In addition to running in real-time, which is a necessity in camera control for visualisation, the camera controller should also count for visibility since the complexity of the geometry means that parts of the geometry might be hidden by other parts.

1.2.3 Interactive Storytelling

Interactive storytelling is “a form of interactive entertainment in which the player plays the role of the protagonist in a dramatically rich environment.” [Crawford, 2004]. A typical interactive storytelling system has a narrator that produces different narratives based on interaction with the player. This provides a rich basis on which to select shots and edits automatically [Christie et al., 2008].

The fact that the narrative unfolds differently based on the interaction with the player imposes the necessity of real-time operation on the camera system of an interactive storytelling system. Additionally, the camera system is required to count for visibility and frame coherency to avoid disorienting the player.

Finally, as a special form of film, principles from cinematography is adopted in interactive storytelling systems. These principles are necessary to better express the meaning of the scenes and thus help the spectator draw conclusions about the plot in general.



Figure 1.2: According to the Rule of Thirds, it is aesthetically better to place the main subject of interest on any one of the four power corners made by two equally spaced horizontal and vertical lines.

1.3 Composition in Camera Control

Another important problem in camera control is that of composition, which forms part of the basis of the contribution of the thesis and needs detailed explanation. *Composition* is “the controlled ordering of the elements in a visual work as the means for achieving clear communication” [Grill and Scanlon, 1990, p. 8]. In other words, composition answers the question of where precisely to place elements in a visual art, and why. Over the time, artists have developed a set of rules that serves as the grammar of composition. Probably the most famous rule in composition is the Rule of Thirds, which proposes that “a useful starting point for any compositional grouping is to place the main subject of interest on any one of the four intersections [called the power corners] made by two equally spaced horizontal and vertical lines” [Ward, 2003, p. 124] (figure 1.2).

The importance of composition lies in that it allows better communication of the message of the picture or drawing, as the picture or drawing is the only source of information available about the content. As for camera control, an additional importance lies in that, as Giors noted (2004) , the camera is the window through which the spectator interacts with the simulated world.

Grill and Scanlon (1990) identified four compositional control types: *graphic controls*, *photographic controls*, *colour controls*, and *total controls*. In the following these controls are explained,

then the chapter is concluded with a brief description of the system and where it fits in the compositional control types.

1.3.1 Graphic Controls

Graphic controls are the basic concepts used in painting, drawing, and some other 2D visual arts to express the interrelationships of the points, lines, and other shapes in the visual art. Though a photograph is usually realised as a three-dimensional scene, it is in essence a two-dimensional image, and thus shares many of the concepts used in these visual arts.

The importance of graphic controls in photography, and in general any 2D visual art, comes from the basic concept of graphics that the human mind seeks to identify patterns to better interpret the meaning of a photograph. Actually, even if there is no pattern in the image, the human mind still tries to identify patterns, which, if they do not exist, means that the image might be wrongly interpreted.

An example of a graphic control concept is that behind the *golden section*. A golden section “is defined as a line that has been divided so that the length of the smaller segment as compared to that of the larger is equal to the length of the larger segment as compared to that of the entire line.” [Grill and Scanlon, 1990, p. 22] Now a good rule in designing a visual art is to have the two adjacent sides of the frame proportion according to the golden section. In addition to having pleasant proportions, the golden section is widely reflected in nature; flowers, the arrangement of branches along the stems of plants, snails, animal skeletons, and even quantum world, all reflect the golden ratio.

1.3.2 Photographic Controls

In addition to the graphic contents of a photograph, there are other features unique to a photograph. These features are produced by the parts and mechanisms of photography of the camera itself: films, lenses, and shutters. For example, a wide open shutter makes the objects in focus sharp, while blurring objects out of focus. This technique can be used to draw attention to the main object.

Photographic controls are very important to a photograph. In fact, as Grill and Scanlon phrased it, “a message presented photographically that could be expressed as well via some other medium—paint, for example—is simply not photographic art.” [Grill and Scanlon, 1990, p. 54]

1.3.3 Colour Controls

A problem in photography arises from the fact that the colours of objects may appear different to the eye than on film. The reason for that is that colours are the eye’s translation of light, so any change in the quantity of light on the camera film means changing the colour itself. This makes colour controls as important to photography as graphic and photographic controls.

Many factors impact the colours of a photograph. The film itself has an impact on the colours and a professional photographer must be familiar with the film to produce the required colour. In addition to that, the photographer might apply colour filters to reflect certain meanings.

1.3.4 Total Controls

Unlike the first three controls, which are mainly concerned with breaking composition into smaller manageable concepts, *total control* is concerned with the integration of the various parts of a photograph into a whole composition. That is, all parts of a photograph support the theme and meaning of the photograph. For example, a small character in a large area might convey the insignificance of the character in the area. For more details, the reader is referred to Grill and Scanlon (1990) .

1.4 Towards Composition in Camera Control

Despite the importance of composition, it has received relatively little attention from researches on camera control, mainly focusing on simple concepts like positioning and framing, as opposed to advanced aesthetic rules. The aim of this research is to produce a camera system that considers concepts from composition in controlling the camera. Specifically, the camera system considers concepts from the first two compositional controls: graphic and photographic controls.

In chapter 2, the literature of camera control in general, and in composition specifically, are reviewed. In chapter 3 the compositional rules that have been considered in the camera system and how their satisfaction is evaluated are discussed. Chapter 4 is dedicated to describing the method used to control the camera.

One of the things this research specifically addresses is the limited number of rules implemented in work on composition. Chapter 5 is dedicated to the presentation of a method for solving this problem by presenting a camera language that can be used to implement different rules based on image processing operators. Finally, chapter 6 concludes the thesis.

Chapter 2

Literature Review

This chapter is devoted to a discussion of the research literature in camera control. The literature of camera control is large, so this chapter concentrates on problems that led to the major developments, then discusses composition, which is the subject of the thesis. For composition, in addition to work in camera control, work in digital photography world is also considered. The chapter concludes with a discussion of the limitations of the previous approaches to composition.

2.1 Classification of Camera Control

Camera control can mainly be classified into *interactive control* and *automatic control*. Interactive control is the control of the camera based on input from a user. For example, in a game, the camera is typically controlled by the player through the keyboard or the mouse. On the other hand, automatic control is control of the camera based on pre-specified rules or animation plot. For example, in a cinematography system, the camera is usually controlled based on the plot of the film.

2.1.1 Interactive Control

The simplest case of interactive camera control is when a user has full direct control on the camera through some input device. Usually, camera parameters, e.g. position, are mapped to the degrees of freedom of the input device. For example, in a typical First Person Shooter (FPS) game, the camera position is mapped to the keyboard arrows and the camera orientation is mapped to the mouse. Ware and Osborne (1990) were among the first to discuss mapping the camera parameters to input devices. They used a 6-dimension input device and identified three possible metaphors:

1. **Eyeball in hand metaphor** The camera is mapped to the input device, such that moving or rotating the device moves or rotates the camera, respectively.
2. **Scene in hand metaphor** The scene (rather than the camera) is mapped to the input device, such that moving or rotating the device moves or rotates the scene, respectively.

3. **Flying vehicle metaphor** The input device is mapped to a virtual vehicle which the user sees the scene through, such that moving or rotating the device changes the positional and rotational velocities of the vehicle.

Each of these mappings has its own advantages and disadvantages. For example, in their experiments, Ware and Osborne found the flying vehicle metaphor to be the best for their maze problem, a problem in which the user is required to locate three areas of detail in a T-shaped hallway [Ware and Osborne, 1990].

The problem with all three metaphors is that they can only be used to explore large spacious areas, and proved to be hard to control while inspecting small objects. For example, approaching an object from a distance with a constant speed causes the object to rapidly become large when the camera is close to it. To solve this problem, Mackinlay et al. (1990) used a logarithmic motion function that moves the camera viewpoint the same relative percentage of distance toward the object in each iteration.

2.1.1.1 Through-the-Lens Camera Control

The main limitation of the methods of [Ware and Osborne, 1990, Mackinlay et al., 1990] is that they control the camera rather than the screen contents, making it unsuitable for many applications, e.g. CAD applications. To solve this problem, Gleicher and Witkin (1992) developed a body of techniques referred to as *through-the-lens* camera control. Rather than controlling the position and orientation of the camera, through-the-lens camera control allows the user to control points on screen and the required camera transformation is found accordingly. Starting with the main camera equation, which projects world points on screen, its derivative is found with respect to the camera parameters (e.g. position). In other words, the relation between the velocity of screen points and the velocity of camera parameters can be found, which, after simplification, has the following form:

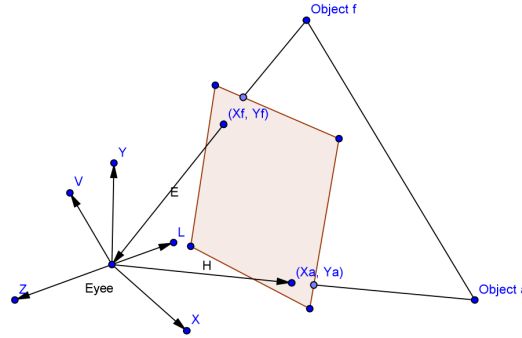
$$\dot{\mathbf{p}} = \mathbf{J}\dot{\mathbf{q}}$$

Having this relation, it is required to find the value of $\dot{\mathbf{q}}$ that set $\dot{\mathbf{p}}$ to a specified $\dot{\mathbf{p}}_0$. This problem is treated as an optimisation problem and the following energy function is formed:

$$E = \frac{(\dot{\mathbf{q}} - \dot{\mathbf{q}}_0) \cdot (\dot{\mathbf{q}}_0 - \dot{\mathbf{q}}_0)}{2}$$

where $\dot{\mathbf{q}}_0$ is the previous velocity of camera parameters. The goal is then to find the value of $\dot{\mathbf{q}}$ that minimises the energy function, for which the method of Lagrange multipliers is used, with the condition that $\dot{\mathbf{p}}_0 = \mathbf{J}\dot{\mathbf{q}}$. This yields, after simplification, the equation

Figure 2.1: Illustrating Blinn's method. Image taken from [Blinn, 1988].



$$\dot{\mathbf{q}} = \dot{\mathbf{q}}_0 + \mathbf{J}^T \lambda$$

which is an ordinary differential equation that can be solved numerically. One method, Euler's method, yields this solution:

$$q(t + \Delta t) = q(t) + \Delta t \dot{\mathbf{q}}(t)$$

This relation gives the new values of the camera parameters given the old parameters.

There are many other methods for interactive camera control. See [Christie et al., 2008] for a detailed discussion.

2.1.2 Automatic Control

Blinn (1988) was one of the researchers to address the problem of automatic camera control. While working at NASA and considering the process of making space movies, he addressed the problem of “where to place the camera and in what direction to point it.” Specifically, he addressed the problem of positioning a spacecraft and a planet on certain positions of the screen. In other words, the goal was to find the position and orientation of the camera such that a spacecraft and/or a planet are placed on certain screen positions. The problem was formulated using vector algebra. The input of the method is the world coordinates of the spacecraft and the planet, their positions on screen, the up vector of the camera, and the camera field of view (figure 2.1). In addition to the standard look-at transformation, Blinn identified three different viewing modes:

1. Given the direction of the camera, position it so that the spacecraft appears at position (X_f, Y_f) on screen and is d units from the camera.
2. Given the position of the camera, direct the camera so that the planet is at position (X_a, Y_a) on screen.

3. Given the required positions on screen of the spacecraft and the planet, position and direct the camera to satisfy them.

Although two big advantages of Blinn’s method are ease of implementation and efficiency of computation, it has the following limitations, which are major problems in camera control:

1. **Occlusion:** The method doesn’t consider other objects in the scene, and thus target objects might be occluded by other objects.
2. **Abstraction:** The method abstract objects as points making the computation inaccurate. For example, placing the object close to screen edge might cause the object to be partially invisible.
3. **Limited applicability:** The method is limited to two targets only.
4. **Inflexibility:** Having to specify exact screen points of the targets makes the algorithm difficult to use, for in camera control the placement of objects is usually specified in ranges rather than exactly, which also adds flexibility to camera placement and orientation.

Each of these problems has received considerable attention from the research community, and there is still no complete solution (i.e. a solution that works independent of the scene and application context) to any of them, but only solutions that work under certain conditions. Some of those solutions are discussed in the following sections.

2.2 Occlusion in Camera Control

2.2.1 Occlusion Detection

Occlusion is one of the fundamental and challenging problems in camera control due to the increasing complexity of the scenes and object models, which limit the available unoccluded view positions satisfying the required shot goals. To make an occlusion-free camera system, two steps are typically required: *occlusion detection* and *occlusion avoidance*.

One of the methods for occlusion detection depends on raycasting. In this method, a ray is cast from the target object to the camera, and its intersection with potential occluders is tested. An intersection with an object means that the object lies between the camera and the target object, resulting in occlusion. To simplify intersection calculation, object scenes are usually approximated with bounding boxes or spheres, resulting in accuracy problems.

Another method for assessing occlusion is based on rendering the scene from the required camera position and testing the colour and depth buffers to examine the existence of other objects (i.e. occluders) between the camera and the target. This method is expensive, since it depends on rendering the whole scene. To optimise it, Halper et al. [Halper and Olivier, 2000, Olivier et al.,

1999] use a bounding sphere test to determine which objects could possibly occlude the target object and render only those objects, rather than the whole scene.

Other methods depend on pre-avoidance of occluded positions by calculating volumes of space which avoids occlusion. For example, Bares et al. (1998) projected the bounding boxes of nearby potential occluders onto a sphere surrounding the target object, then converted those projections to global spherical coordinate system. When those projections are negated, they result in occlusion free regions.

Christie et al. (2005) followed a similar approach to that of Bares et al. and built volumes of partial and total occlusion around two characters (see figure 2.2a). A camera placed in one side of the cylinder of total occlusion will not be able to show the object on the other side. Similarly, a camera placed in one side of the cone of partial occlusion will be able to only partially show the object on the other side. Although efficient, the shortcoming of this approach is being limited to two objects only.

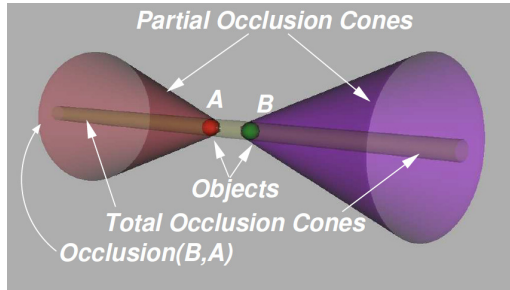
2.2.2 Occlusion Avoidance

The second step in making occlusion-free camera system is occlusion avoidance. Given a camera configuration satisfying the required goals but having the occlusion problem, the goal of this step is to find the closest camera configuration that removes occlusion.

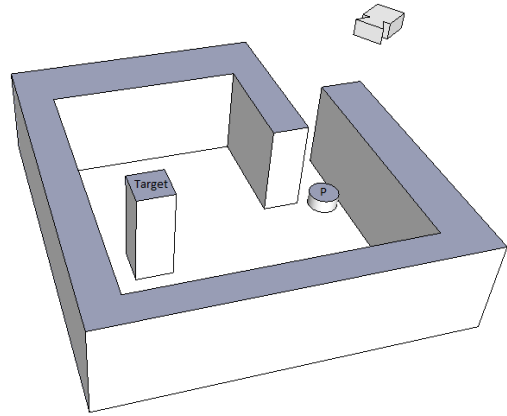
Philips et al. (1992) rendered the scene on a hemicube centred around the target object and has its opening towards the camera. If the centre of the image (which corresponds to the camera location) contains pixel data, it means there is an occluder between the camera and the target object, and the system looks around the centre of the image for an empty pixel to place the camera at. However, Halper et al. (2001) pointed out that since this method does not check depth information, it would fail, in a configuration such as that shown in figure 2.2b, to find the position P since the wall will occupy most of the hemicube map.

To solve this problem, Halper et al. (2001) checked for depth information while rendering from the target object to the camera. The first step was to render the scene with depth buffer enabled and colour buffer disabled, resulting in a black image with depth data corresponding to the objects in the scene as seen from the camera to the target object. Then, some objects (e.g. spheres) were rendered on the target image using bright colour on favourable positions. The result was that those areas where an occluder exists would have low depth values, and thus nothing will be rendered, keeping those areas black. The system could then process the image and search for the light areas and choose the lightest position nearest to the centre of the image as the non-occluded camera position.

Other methods of occlusion avoidance depends on formulating occlusion as a property to solve for and use constraint satisfaction or optimisation to avoid occlusion. For example, Bares et al. (2000b) formulated the property **OBJ_OCCLUSION_MINIMIZE** that requires that no more than a specified minimum fraction of object be occluded, where the occlusion fraction is the ratio



(a) Volumes of partial and total occlusion. A camera placed in one side of the cone of partial occlusion will be able to only of the cylinder of total occlusion will not be able to shoot the object on other side. Image taken from [Christie and Normand, 2005].



(b) Halper et al. [Halper et al., 2001] pointed out that in a configuration like this, the hemicube method of Phillips et al. [Phillips et al., 1992] would fail to find the position P to avoid occlusion while shooting the target. Image taken from [Halper et al., 2001].

of occluded pixels to total number of pixels. The occlusion property, along with other screen and camera properties, were solved as a whole using constraint satisfaction techniques. Similarly, Olivier et al. (1999) formulated occlusion as a property, but used optimisation (genetic algorithms) rather than constraint satisfaction.

2.3 Abstraction

Abstracting objects as points or simple geometric objects simplify computations and usually produce inaccuracy. For example, in the raycasting method of occlusion detection mentioned in the previous section, objects are abstracted as points. This overly simplifies the computation and might produce problems such as that shown in figure 1.1b. To mitigate this problem, the raycasting method can be repeated on different points of the target object. For example, in the case of humanoid or similarly shaped creatures, three positions are usually used: head, torso, and feet [Haigh-Hutchinson, 2009]. This mitigates the abstraction problem, but does not remove it.

Similarly, Halper et al. (2001) repeated the same process (of rendering with depth buffer enabled and colour buffer disabled, etc.) from several different points on the target objects to mitigate the problem of abstraction. Again, this mitigates the problem, but does not remove it. Additionally, each additional point means additional rendering, which makes the method slower the more accurate it is. For example, if we follow Haigh-Hutchinson's suggestion [Haigh-Hutchinson, 2009] of using three points, on head, on torso, and on feet, the scene need to be rendered four times rather than just once.

Another method depends on rendering to produce accurate results [Bares et al., 2000a]. The target object is rendered on a back buffer with stencil mask enabled. Then potential occluders are

rendered using unique colour codes. The result is that non-zero pixels having values different than those of the target object are occluded pixels, enabling the accurate calculation of the fraction of object in occlusion.

2.4 Target Objects Limit in Camera Control

The problem of camera control imposes a limitation on the number of target objects the camera can actually show in a single shot. There are two different reasons for this. The first is inherent in the nature of the problem even outside the virtual world. For example, though it is possible to widen the field of view to include two characters far apart in the same shot, this usually distorts the contents of the shot. In such cases, cinematographers usually bring the characters close to each other while shooting the conversation (i.e. stage the scene differently).

The second reason lies in the complexity of the computation required to combine multiple objects in a shot. Additionally, such computations are usually expensive, limiting their application in real-time applications. Generally, the more targets to consider, the slower the method will be.

Many applications (e.g. third-person shooter games) are centred around one or two characters, for which Blinn's method (1988) is suitable. However, other applications (e.g. films) usually require many targets in the same shot. In such cases, many applications depend on optimisation methods. For example, in their CamPlan system, Olivier et al. (1999) projected objects on screen and used genetic algorithms to optimise for the camera parameters. Similarly, Burelli et al. (2008) used Particle Swarm Optimisation (PSO) to solve for the camera parameters.

Other methods separate the problem into two steps. For example, Lino et al. (2010) built *visibility volumes* around one or two objects, in which the camera can show the target objects without occlusion, and searched inside those volumes to account for other objects.

2.5 Flexibility Problem in Camera Control

The problem of camera control requires two classes of flexibility, flexibility in expressiveness and flexibility in the generated solutions. The former is required because it gives more flexibility in finding solutions, and it is usually justified because slight changes in the positions of screen elements have almost no noticeable effect. The second is required because there are usually many solutions satisfying the requirements, and generating only one solution means removing other feasible positions that can be more suitable in cases like animations where a smooth coherent path is required.

As for the inflexibility in expressiveness, many approaches avoid this problem by specifying a range of target positions/sizes/etc rather than exact values. For example, in their camera system, Burelli et al. (2008) formulated the “**Object Position in Frame**” property, which required “a specified fraction $f \in [0, 1]$ of the object to lie inside a given rectangular subregion of the image”.

Similarly, Olivier et al. formulated properties that require objects to lie between two lines, objects to lie to the left of other objects, etc.

As for the inflexibility in generated solutions, one of the solutions was proposed by Christie et al. (2005). They proposed partitioning space around important characters into volumes sharing certain visual properties, each called a *semantic volume*. More formally, a semantic volume is defined “as a volume of possible camera locations that give rise to qualitatively equivalent shots with respect to cinematographic properties.”

Christie et al. realised the following properties in their system: projection size on screen (close-up, medium etc.), orientation (in front of object, left-profile of object, etc.), occlusion (whether objects occlude each other), and framing (whether the object is inside, partially-inside, or outside the screen). The space around the main objects is divided into volumes satisfying these properties and each volume is tagged with the properties it satisfies. Moreover, if a volume satisfies more than one property, the volume will be tagged with all those properties.

According to the user requests, a volume or a set of volumes are retrieved. Since the volume itself does not specify a certain camera location, another algorithm should be run inside the volume to choose the best position according to some criteria.

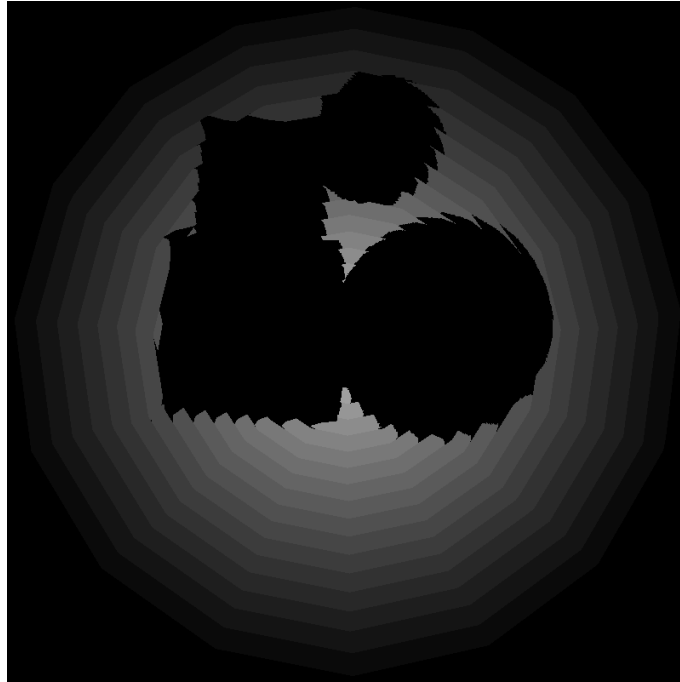
2.6 Camera Control in Applications

2.6.1 Camera Control in Games

Halper et al. (2001) developed an automatic camera engine for computer games. A constraint-based approach was used to specify the position of the camera relative to the target object. The constraints specified the camera height, camera angles relative to the objects, visibility, and object size and position on screen. To avoid the jumpiness that results from a purely reactive application of the constraints, the camera system first planned the next camera state based on the current camera state, then, with relaxation parameters specified for constraints, the camera system tried to find the best approximation of the result that preserves frame coherency.

To address the visibility of the object, a visibility map was created by rendering from the object to the camera, then the empty point nearest to the centre of the visibility map was found, on which depth-buffer inversion was applied to get a non-occluded camera position (figure 2.2). Although the method is efficient in computation, objects are abstracted as points making the computation inaccurate. Also, it is difficult to extend the algorithm to multiple objects, as that requires using shadow techniques which makes the computation much heavier. Finally, not knowing in advance whether a certain camera position results in occlusion or not (because occlusion is detected lastly using reversed rendering) means that the camera solver might miss other less satisfactory but non-occluded camera positions.

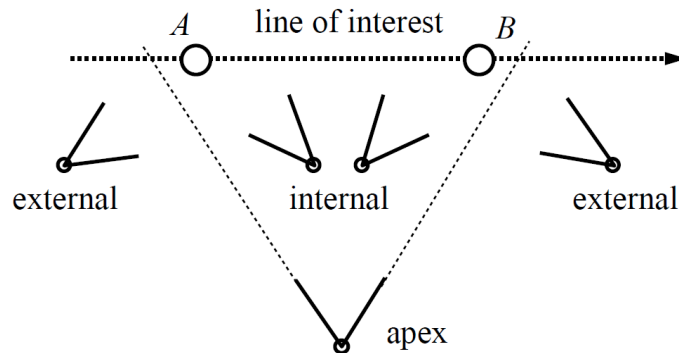
Figure 2.2: Halper et al. (2001) renders depth information from the camera towards the target, then render a set of objects of preferred locations in the scene with depth test enabled. The result is that occluded areas will have black colours, while non-occluded areas will have white colour with varying brightness according to preference, among which the brightest point is chosen, and depth-buffer inversion is used to get a non-occluded camera position. The image is taken from [Halper et al., 2001]



2.6.2 Camera Control in Cinematography

The work of Christianson et al. (1996) was one of the first to approach the problem of cinematography. They developed the Declarative Camera Control Language (DCCL) to enable the definition of film idioms (see section 1.1.5) based on simple fragments combined together. A fragment can be briefly defined as the behaviour of the camera during a certain period. They supported basic fragments like *static*, in which the camera is fixed to a certain position and direction, *panning*, in which the camera is fixed to certain position but rotates to track a character, and so on. The camera position and orientation were calculated using Blinn's method (1988). Each scene was specified with a set of idioms that can be used to shoot the scene. The sequence of scenes constituted the film. To generate the sequence of scenes, a sequence planner was implemented for their specific plot, which was "chase and capture interactions." The sequence planner got the animation trace, which contained information about the actors and their activities, and partitioned the animation into scenes accordingly. Then, according to the activities performed by the actor, the scenes were partitioned into idioms. The partitioning of sequences into scenes, scenes into idioms, and idioms into fragments, constituted what they termed *film tree*. The final step was to run a heuristic evaluator to choose the best idioms to shoot the film. The evaluator aimed to maintain smooth transition, eliminate fragments which cause the camera to cross the line of interest, eliminate very short or

Figure 2.3: Different camera placements for a two-character conversation. This image is taken from [He et al., 1996], which is adapted from [Arijon, 1976].



very long shots, and eliminate fragments in which the camera pans backwards.

Another work on cinematography was *Virtual Cinematographer* by He et al. (1996). It supported camera modules, each of which automatically configured the camera according to cinematography rules. Examples of these camera modules are apex, external, internal (figure 2.3). They implemented cinematographic idioms through the use of finite state machines (FSM). Each state in an FSM invoked a particular camera module and had a list of conditions to move to another state. The use of FSM gave more flexibility in making the camera react to events in the scene. For example, it was possible to change the camera to show a character when the character started speaking. There are, however, shortcomings in their approach. The limited conditions for traversal of the FSM made it difficult to incorporate the editing process (e.g. where and when to cut). Also, the camera modules returned single solutions which made it impossible to apply composition rules, as the latter requires flexibility in moving and orienting the camera.

Lino et al. built a cinematography system based on the approach of semantic volumes (described in section 2.5). Semantic volumes were defined around one or two characters according to shot styles derived from cinematographic conventions (figure 2.4). For example, an **external close-up** semantic volume around two characters is a volume from which it is possible to shoot the two characters from above the shoulder of one of them (i.e. external) and have only the heads of the characters appear in the shot (i.e. close-up). To avoid occlusion, the volume in which a camera can show a character(s) without occlusion was computed. This volume was termed the *visibility volume*. Intersecting semantic volumes with visibility volumes produced what they termed *director volumes*, in which a good camera configuration was searched for.

2.7 Composition in Camera Control

In chapter 1 composition was defined to be “the controlled ordering of the elements in a visual work as the means for achieving clear communication” [Grill and Scanlon, 1990, p. 8]. In a sense, all the problems in camera control can be considered composition problems, for the simple reason

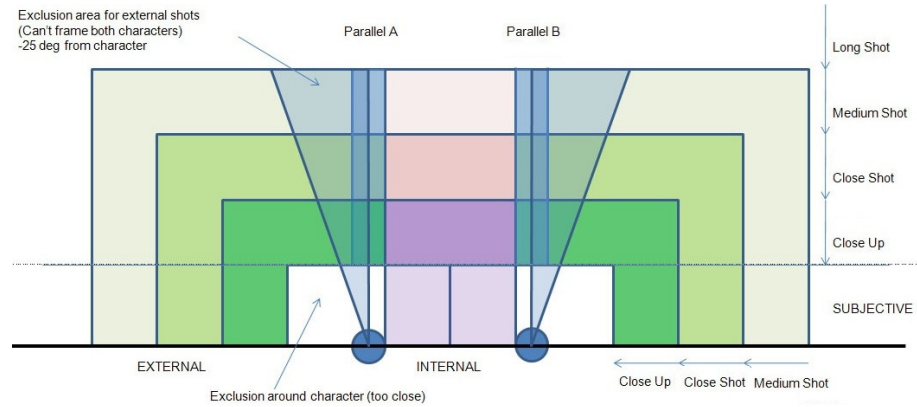


Figure 2.4: The semantic volumes around two characters in the cinematography system of Lino et al. [Lino et al., 2010], from which this image is taken.

that the goal of camera control is to achieve clear communication by wisely placing elements on screen. However, the problem of composition in camera control is usually concerned with the placement of multiple objects on screen and their absolute and relative positions, rather than the placement of one or two objects on certain positions at screen and tracking them (e.g. games).

One of the earliest work on composition problem in camera control was CamPlan [Olivier et al., 1999, Halper and Olivier, 2000], a camera planning agent utilising a set of relative and absolute composition properties to be applied on screen objects. Examples of such properties are: *HorizSize*, which specifies the horizontal extent of an element, *BetweenX*, which requires the screen coordinate of a screen element to lie within a certain minimum and maximum. Evaluation functions were assigned for those properties so that an optimisation method could be applied. Camera parameters (position, rotation angles, and field of view) were encoded in a gene and genetic algorithms were applied to find the best possible camera position. The main shortcoming of CamPlan is that it *lacks support of advanced composition rules from photography and cinematography* such as the rule of thirds.

Burelli et al. (2008) developed a system similar to CamPlan, but realised two different types of properties: image properties, and camera-related properties. Like CamPlan, image properties were concerned with the positions of elements of screen. However, camera-related properties were also used to bind the camera to certain volumes of space. For example, the *Camera Above Plane* property “requires the camera to lie above a given plane in 3D space.” The different camera-related properties were used to generate volumes of feasible camera positions, inside which they used Particle Swarm Optimisation to find the best possible camera configuration. Like CamPlan, this approach *lacks support of advanced composition rules*.

Motivated by the assumption that a human photographer starts with an initial guess of the camera position then improves the resulting photograph by repeated fine tuning of position and zooming, Bares (2006) created a photographic composition assistant for virtual camera systems. The system improved composition by applying transformations on the camera to shift and resize

screen elements to achieve certain rules like the rule of thirds. The main limitation of this approach is that it can only be used to *improve* camera configurations rather than *find* camera configurations. In the context of the thesis, this problem is called *the limited applicability* problem, which is defined to be either the limited results of the method (e.g. methods that results in partial solution, or can be applied to few objects) or the inability to apply the method to computer graphics.

Another limitation that exists in many other camera systems is the *inaccuracy resulting from using approximate geometric shapes like bounding boxes or spheres to represent objects in scene*. In some cases such as when approximating a box-shaped object, the inaccuracy is negligible. But some other cases might produce serious accuracy problems, like having a long diagonal rod, as the bounding box for such case is a very rough approximation. As a step towards solving this problem, Ranon et al. (2010) developed a system to accurately measure the satisfaction of visual properties. They developed a language that enabled the definition of different properties depending on *rendering operators*, which return the set of pixels occupied by a certain screen element after the scene is rendered from a certain camera configuration. This set of pixels can be processed with different image processing functions to measure the satisfaction of visual properties. For example, the function **Overlap** takes two pixel sets and return the overlapping pixels. The main shortcoming of their system is that it only evaluates the satisfaction of properties, rather than finding camera configurations satisfying them.

In digital photography world, Banerjee et al. (2004) described a method to apply the Rule of Thirds to photographs by shifting the main subject(s) in focus so that it lies on the closest power corner. The main subject(s) was(were) detected using a wide shutter aperture that blurred objects out of focus. This approach has the *limited applicability* problem in that it can only be applied to one object and one rule, and it cannot be applied in computer graphics. Similarly, Gooch et al. (2001) applied the Rule of Thirds on a 3D object by projecting its silhouette on screen and matching it with a template. Again, the method is limited to one object only, and thus has the *limited applicability* problem.

Finally, Liu et al. (2010) addressed the problem of composition in digital cameras using cropping and retargeting. They started by extracting the salient regions and prominent lines using image processing techniques. Having those, Particle Swarm Optimisation [Kennedy and Eberhart, 1995] was applied to find the coordinates of the cropping that maximise the aesthetic value of an image based on *rule of thirds*, *diagonal dominance*, and *visual balance*. Finally, the resulting cropped image was retargeted to the dimensions of the original image. Though it addresses advanced composition, the method has the two parts of the *limited applicability* problem: it always apply the same rules on the same objects, and it cannot be applied to computer graphics.

2.8 Conclusion

In this chapter, the work on camera control that led to the major developments was discussed, followed by a discussion of the important work on composition, which pointed out the following shortcomings of previous work:

1. The lack of support of advanced composition rules (e.g. rule of thirds) in camera control systems.
2. The approximation of complicated models with approximate geometric shapes, which makes the solution inaccurate.
3. The limited applicability of many methods to few objects and few rules, or the inability to apply other methods (e.g. methods of digital cameras world) to computer graphics.

This research is an attempt to solve these shortcomings. The first problem is the main goal of this thesis, and is considered in chapter 3, along with the second problem. The third problem is considered in chapter 4. Finally, another dimension of the limited applicability is considered in chapter 5, for which a language for camera composition is introduced.

Chapter 3

Formulating the Composition Rules

Chapter 2 discussed the important work on camera control, and concluded with the discussion of the limitation of the current camera control. One important limitation is the lack of advanced composition rules in previous work on composition in camera control. To address this apparent shortfall in functionality, this chapter describes a set of basic and advanced rules of composition and their implementation in an offline camera control system. The rules selected correspond to graphic and photographic controls of the four compositional controls (see section 1.3). The first is concerned about the basic graphic contents of a shot, while the second is concerned about properties realized by controlling the components of the camera, namely, the film, the lens, and the shutter.

3.1 Selecting Rules for Implementation

As a first step towards advanced composition in camera control, some of the most important rules in the literature of photography and cinematography were explored and some of them were selected for implementation. The motivation for selecting these rules is that they have clear impact on the results, as opposed to more delicate rules which might need a professional to perceive. Here is a listing of the selected rules:

1. **Rule of Thirds** [Liu et al., 2010, Banerjee and Evans, 2004, Bares, 2006, Gooch et al., 2001, Byers et al., 2004]: Proposes that “a useful starting point for any compositional grouping is to place the main subject of interest on any one of the four intersections [called *power corners*] made by two equally spaced horizontal and vertical lines” [Ward, 2003, p. 124]. It also proposes that prominent lines of the shot should be aligned with the horizontal and vertical lines [Grill and Scanlon, 1990]. See figure 3.1a for an example.
2. **Diagonal Dominance** [Liu et al., 2010]: Proposes that “diagonal arrangements of lines in a composition produces greater impression of vitality than either vertical or horizontal lines” [Ward, 2003]. For example, in figure 3.1b, the table is placed along the diagonal of the frame.



Figure 3.1: Images illustrating composition rules.

3. **Visual Balance** [Lok et al., 2004, Liu et al., 2010, Bares, 2006]: States that for an equilibrium state to be achieved, the visual elements should be distributed over the frame [Ward, 2003]. For example, in figure 3.1c, the poster in the top-left corner balances the weight of the man, the bottle, the cups, and other elements.
4. **Depth of Field**: This rule controls the parameters of camera lens such that an object is in focus, and other background and foreground elements are out of focus. This rule is used to draw attention to the main subject of a scene (figure 3.1d).

Besides the advanced composition rules, it is also important to have a set of basic rules to help in controlling some of the elements of the frame. For example, it might be desired to show a river across the diagonal of the frame yet require that some animals be placed near the river to the top-left corner of the frame. The diagonal dominance rule can be applied on the river to assure that it is shown diagonally but the other advanced rules cannot be of any use for placing the animal in the top-left corner of the frame. For this, the following basic composition rules are implemented as well:

1. **Framing Rule** [Olivier et al., 1999, Bares et al., 2000a, Burelli et al., 2008, Lino et al., 2010]: Specifies that the frame surrounding a screen element should not go beyond the specified frame. This rule is useful when an element needs to be placed in a certain region of the frame. See figure 3.1e.
2. **Visibility Rule** [Olivier et al., 1999, Burelli et al., 2008, Bares et al., 2000a]: Specifies that a minimum/maximum percentage of an element should be visible. For example, a case like that of figure 3.1f in which showing a character causes another character to be partially in view can be avoided by applying this rule on the character to the right.
3. **Position Rule** [Olivier et al., 1999, Burelli et al., 2008]: Specifies that the centre of mass of an element should be placed on a certain position of the screen.
4. **Size Rule** [Olivier et al., 1999, Burelli et al., 2008, Bares et al., 2000a]: Specifies that the size of a certain element should not be smaller than a certain minimum or larger than a certain maximum. This rule is mainly useful to control the size of an element to ensure its size reflects its importance on the frame.

Each of the previous 8 rules take an object(s) as a parameter and some other parameters depending on the rule's requirements. The reader is referred to chapter 4 for practical examples on the use of these rules.

3.2 Rating Rules

To simplify the implementation of the camera system, many applications depend on methods specific to the application itself, i.e. build the camera system taking into account the scene geometry and the expected activity and interaction between different scene objects. However, the camera system presented in this thesis is scene-independent, with the ability to impose many rules on many objects making the problem strongly non-linear. This, along with the aim of producing aesthetically-maximal results, suggests using optimisation to get the best possible camera configuration, and this in turn requires shots to be rated to evaluate the objective function of the optimisation method.

3.2.1 Geometric Approach vs. Rendering Approach

To measure the satisfaction of composition rules for a certain camera configuration, scene objects must be projected on screen. There are generally two approaches to projection, *geometric approach* and *rendering approach*. In the geometric approach scene objects are approximated using approximate geometric shapes such as bounding boxes or spheres, which are then projected on screen using simple closed-form mathematical equations. The advantage of the geometric approach is the efficiency of computation. However, it has the disadvantage of being approximate, which can result in very inaccurate results in some cases (e.g. a long diagonal rod).

On the other hand, the rendering approach relies on rendering the scene on an image, then using image processing techniques to find the extents of the object. To avoid extra computation, the scene might be rendered partially, so that only the objects under considerations are rendered. In contrast to the geometric approach, the advantage of the rendering approach is the accuracy, while the disadvantage is the inefficiency of the computation.

Many approaches to camera control depended on geometric approach. In this camera system, however, and since composition is about the precise placement of elements on screen, rendering approach is used.

3.2.2 Visibility Issue

An important issue to consider in the rendering method is that rendering the scene as it is tells nothing about the parts of an object which are out of view, making the rating of some rules incorrect. For example, the visibility rule might be applied on an object to make the object fully visible. To rate the visibility rule, the camera system needs to know the parts of objects which are visible and those which are invisible. Unfortunately, the image only contains those pixels of an object which are in the frame.

To solve this problem the camera system uses a field of view wider than the original field of view such that the original view covers only the rectangle having corners (25%, 25%) and (75%, 75%), rather than the whole screen. This is implemented through a vertex program (see appendix B) that brings the pixels resulting from rendering towards the centre of the screen such that the horizontal and vertical distances to the centre of the screen are half their original values. This way the camera system knows which parts of objects are visible and which are invisible. This is illustrated in figure 3.2b, in which the white rectangle represents the separator between the visible and invisible areas.

3.2.3 Silhouette Rendering

Since composition rules apply to certain objects only, which are termed *ruled objects*, the camera system only render those objects and process the resulting image for rules rating. Having only ruled objects, the next step is to extract the extents of objects, for which edge detection algorithms

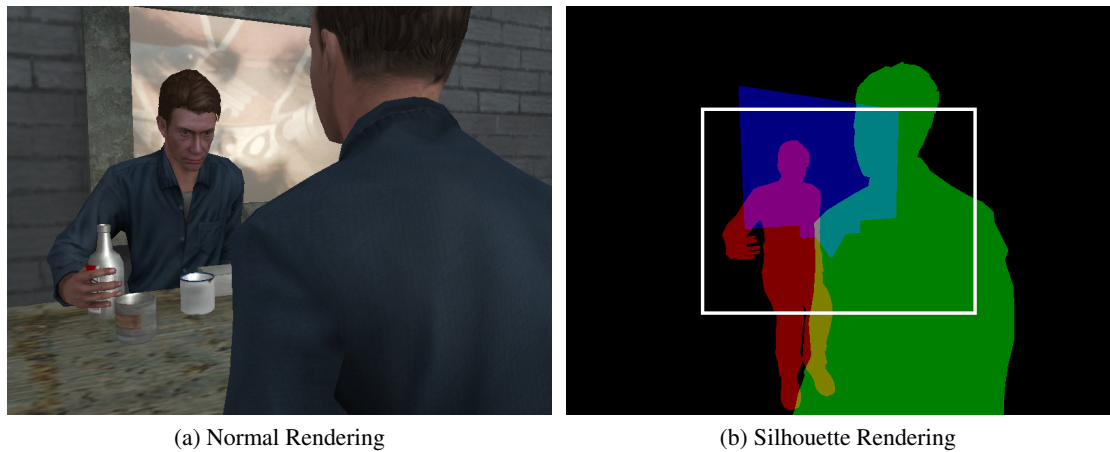


Figure 3.2: To make image processing easier, the camera system only render ruled objects and use different colour for each object. Furthermore, it brings the pixels resulting from rendering towards the centre of the image such that the horizontal and vertical distances of the pixel to the centre of the image are half their original values. This way the camera system can process pixels which are originally out of view.

can be used. However, one problem with edge detection is its cost and probable inaccuracy. Since for the rules selected for implementation the most important aspect is the region occupied by an object rather than its colours, these problems can safely be avoided by rendering object silhouettes only, and assigning a different colour for each object. This is done by using a fragment program (see appendix B). The problem then comes down to comparing each pixel in the rendered image against an object colour. One problem with this approach is that when multiple objects share the same region on the frame, their pixels will overwrite each other according to which one is rendered first. To solve this problem the colours of the objects are restricted such that their integer representations are powers of 2, then blending with addition function is used while rendering the object. This way the same pixel can be occupied by as many as 32 objects for 32-bit rendering. See figure 3.2a and figure 3.2b.

Depending on blending with addition raises another problem which cause objects to fill the wrong bits of the pixel, which is when a certain object render to the same pixel more than once (e.g. a cube which has its back face rendered before the front face). To solve this last problem, the not-equal depth test function is used with distinct depth value manually specified for each object. This way each object can render to the same pixel only once and the problem is solved.

After applying the fragment program, and the vertex program for solving the visibility issue mentioned in the previous section, the result is an image similar to the one in figure 3.2b. Then simple image processing techniques are used to rate the camera configuration producing the image. The details of the image processing techniques are discussed in section 3.2.5.

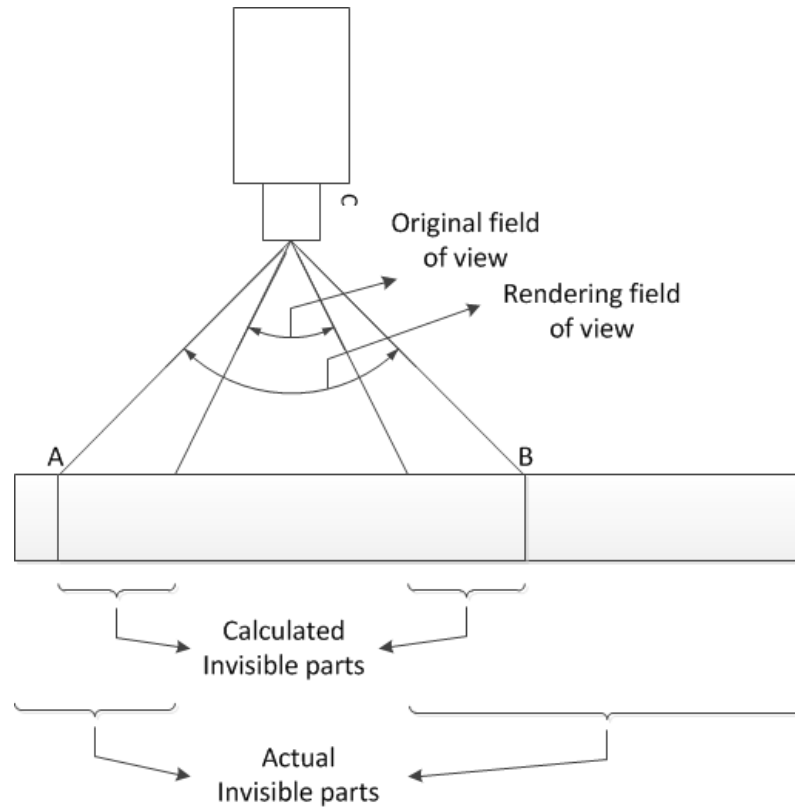


Figure 3.3: The camera system uses a field of view wider than the original field of view so that it can process pixels which are out of view. However, the configuration in this figure shows an example of a case in which the object is large enough to be fully contained in the new field of view. In such cases, the calculated visibility is larger than the actual visibility.

3.2.4 Rating Functions

To simplify the calculation of rule rating, each rule is divided into a set of factors determining its rating. For example, the rating of the diagonal dominance rule is determined by two factors: the angle between the prominent line of an object and the diagonal line of the screen and the distance between the prominent line and the diagonal line (figure 3.4). For any rule, the camera system rates each of its factors, then aggregate the values to get the final rating of a single rule.

To get the best results from the optimisation method used (discussed in chapter 4), the following criteria for the rating of each factor is suggested:

1. While the function must evaluate to 1 when the factor is fully satisfied, it must not drop to 0, otherwise the method will be merely an undirected random search. However, after some point, which is termed the *drop-off point*, the function should drop heavily to indicate the dissatisfaction of the factor.
2. The function must have some tolerance near the best value, at which the function will still have the value 1. This gives more flexibility to the solver.

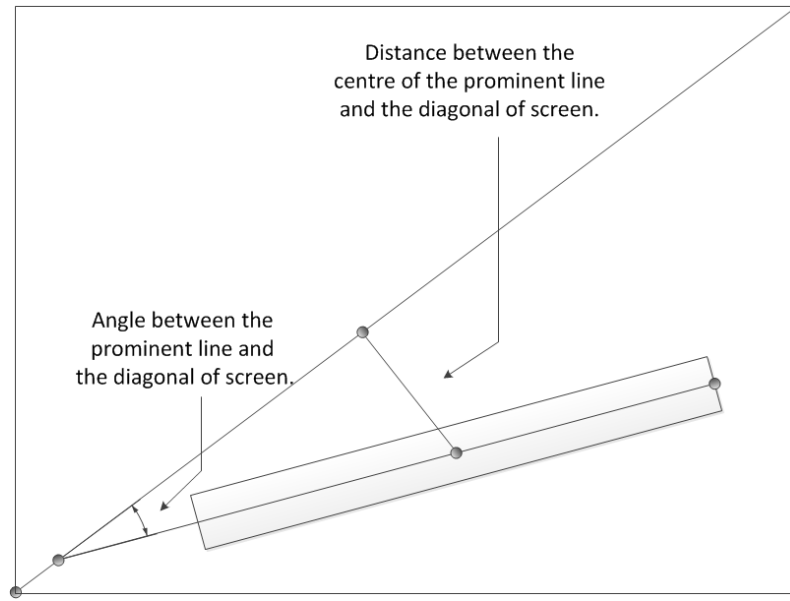


Figure 3.4: Illustrating the two factors determining the rating of the diagonal dominance rule.

A good match for these criteria is Gaussian function. For each factor, a best, tolerance, and drop-off values are decided, and then Gaussian function is used as follows:

$$FR = e^{(\frac{\Delta - \Delta_t}{\Delta_{do} - \Delta_t})^2} \quad (3.1)$$

where FR is factor rating and

Δ :the difference between the current value of the factor and the best value

Δ_t :the tolerance of the factor

Δ_{do} :the difference between the drop-off value and the best value.

Having the rating of each factor of a rule, geometric mean is used to find rule rating:

$$Rule\ Rating = \sqrt[n]{\prod_{i=1}^n FR_i} \quad (3.2)$$

The reason behind using geometric mean rather than arithmetic mean is that geometric mean has the property of reducing to zero when one of the values reduces to zero, which is an important property here since the satisfaction of all factors is required to satisfy the rule.

The data in table 3.1 gives the parameters of the different factors of all the composition rules supported in the camera system. Table 3.2 lists some of the symbols used in table 3.1. It should be noted that the *Rule of Thirds* is separated into two rules, one for the *power corners* and the other for horizontal and vertical lines. This is because they usually apply to different elements of the screen.

Factor	Best Value	Δ_t	Δ_{do}	Explanation
Rule of Thirds (Corners)				
Horz. Distance to Corner	0	HD/20	HD	The horizontal distance between the centre of mass of the object and the closest corner.
Vert. Distance to Corner	0	HD/20	HD	The vertical distance between the centre of mass of the object and the closest corner.
Rule of Thirds (Lines)				
Line Angle	0 or 90	5	30	The angle of the prominent line of the object.
Diagonal Dominance				
Line Angle	45	15	30	The angle between the prominent line of the object and the diagonal lines.
Line Distance	0	0.25	1	The distance between the prominent line of the object to the diagonal lines.
Visual Balance				
Horz. Centre	0	0.1	0.5	The horizontal component of the centre of mass of all the objects of the rule.
Vert. Centre	0	0.1	0.5	The vertical component of the centre of mass of all the objects of the rule.
Framing Rule				
Left Outside	0	5% FW	25% FW	The width of the part of the object which is beyond the left border of the framing specified by the rule.
Bottom Outside	0	5% FH	25% FH	The height of the part of the object which is beyond the lower border of the framing specified by the rule.
Top Outside	0	5% FH	25% FH	The height of the part of the object which is beyond the upper border of the framing specified by the rule.
Right Outside	0	5% FW	25% FW	The width of the part of the object which is beyond the right border of the framing specified by the rule.
Visibility Rule				
Beyond Min. Horz. Visibility	0	5% AOW	25% AOW	The amount the horizontal visibility of the object is beyond the minimum horizontal visibility.
Beyond Min. Vert. Visibility	0	5% AOH	25% AOH	The amount the vertical visibility of the object is beyond the minimum vertical visibility.
Beyond Max. Horz. Visibility	0	5% AOW	25% AOW	The amount the horizontal visibility of the object is beyond the maximum horizontal visibility.
Beyond Max. Vert. Visibility	0	5% AOH	25% AOH	The amount the vertical visibility of the object is beyond the maximum vertical visibility.
Position Rule				
Horz. Distance	0	0.01	0.25	The horizontal distance between the centre of mass of the object and the position specified by the rule.
Vert. Distance	0	0.01	0.25	The vertical distance between the centre of mass of the object and the position specified by the rule.
Size Rule				
Beyond Min. Width	0	5% AOW	25% AOW	The amount the width of the object is beyond the minimum width.
Beyond Min. Height	0	5% AOH	25% AOH	The amount the height of the object is beyond the minimum height.
Beyond Max. Width	0	5% AOW	25% AOW	The amount the width of the object is beyond the maximum width.
Beyond Max. Height	0	5% AOH	25% AOH	The amount the height of the object is beyond the maximum height.

Table 3.1: The factors the camera solver depends on to rate the rules.

Symbol	Description
HD	Half the distance between the power corners (i.e. 0.33333)
FW	Width of the frame used by the framing rule.
FH	Height of the frame used by the framing rule.
AOW	Average width of the projection on screen of the object being considered by a rule.
AOH	Average object of the projection on screen of the object being considered by a rule.

Table 3.2: Symbols and abbreviations used in factors calculation.

3.2.5 Shot Processing

To rate the satisfaction of a rule the camera system needs to process the image resulting from silhouette rendering (section 3.2.3). The rating of *boundary rule* and *size rule* depend on the frame surrounding the object on screen which can be easily found. The rating of the *visibility rule* depend on the number of pixels in the visible and invisible areas which is also straightforward to calculate.

For *visual balance* and the power corners in the *Rule of Thirds*, the rating depends on the centroid of the object, which can be calculated using the equation:

$$Centroid = (\frac{\sum_{i=1}^n x_i}{n}, \frac{\sum_{i=1}^n y_i}{n}) \quad (3.3)$$

where

(x_i, y_i) :the coordinate of the i th pixel occupied by the object

n :the total number of pixels occupied by the object.

Finally, for *diagonal dominance* and the horizontal and vertical lines in the *Rule of Thirds*, the rating depends on the prominent line of the ruled object. To find the prominent line of an object linear regression is applied on the pixels of the object to find the best fitting line, i.e. the line having the least distance to all the pixels. The standard method for finding the linear regression of a set of points is to minimise the vertical distance between the line and the points. This is mainly useful if the points represent the value a function with respect to a variable, and the aim of the regression is to find the line that minimises the error represented by the vertical distances between the line and the points. However, in the case of finding the prominent line, we want to find a line that fits best rather than a line that minimises the error so a modified linear regression called *perpendicular linear regression* [Weisstein, 2010] is used. The method starts by finding the centroid of all the points then finding the angle of the line passing through the centroid which minimises the perpendicular distance. The angle is found according to the equation:

$$\tan(\theta) = -\frac{A}{2} \pm \sqrt{(\frac{A}{2})^2 - 1} \quad (3.4)$$

where

$$A = \frac{\sum_{i=1}^n x_i^2 - \sum_{i=1}^n y_i^2}{\sum_{i=1}^n x_i y_i} \quad (3.5)$$

Equation 3.4 results in two angles, one of them for the line with the maximum distance to the points and the other is for the minimum distance which is what is needed here. Figure 3.5a

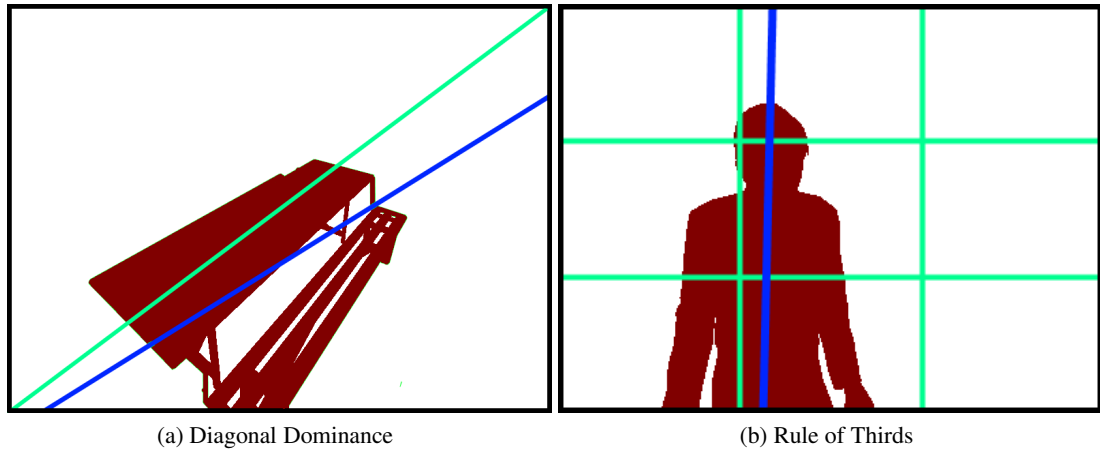


Figure 3.5: Illustrating how the prominent line of an object is found using perpendicular linear regression. The line in blue is the prominent line extracted from the object in dark red.

illustrates finding the prominent line of the table in figure 3.1b. Figure 3.5b illustrates the same concept but for a character to be positioned according to *Rule of Thirds*.

3.2.6 Rule Modules

Each of the implemented rules is encapsulated in a module that serves as the implementer of that rule. The function of each module is to rate the satisfaction of a rule for a certain camera configuration. To rate the satisfaction, the rule module need to connect to the image processor module. The image processor module on the other hand needs to connect to the renderer module, which is responsible for rendering the scene to an offline texture. This is illustrated in figure 3.6.

The rule module itself is triggered by the camera system inside the optimisation loop. Inside each loop, the rule modules of the used rules are sent the camera configuration, then it will respond with the rating of the satisfaction of the rule for the given camera configuration. This will be further detailed in chapter 4.

3.3 Texture Resolution

An important point to consider when rendering the scene on an offline texture is the size of the texture. If the system uses the original size of the rendering, e.g. 800×600 , the transfer from GPU to CPU memory will take a long time, as well as the image processing after the transfer. On the other hand, if the systems use a low resolution texture, it will run into inaccuracy problems, which is one of the problems we wanted to avoid by not using the geometrical approach. This raises the question of what is the least resolution to use, yet avoid running into noticeable accuracy problems.

To answer this question, the relation between the resolution and accuracy should be understood. It is derived as follows: using a texture smaller than the original screen means that many

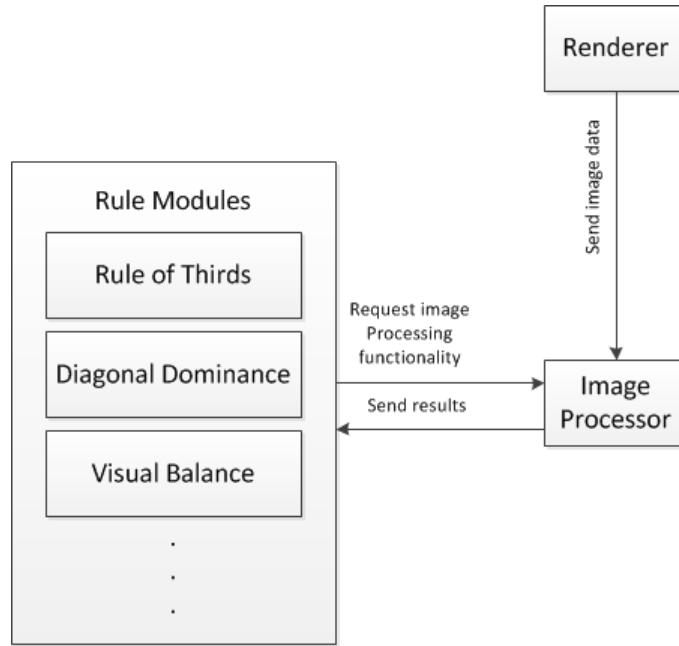


Figure 3.6: The connection between the different module in the camera system.

pixels in the screen will share the same pixel on the texture. So for a screen having a width W , and a texture having a width w , the number of pixels in the screen sharing the same pixel in the texture is W/w . For example, if the width of the texture is 128, then $800/128 = 6.25$ screen pixels will share the same texture pixel. That is, the average inaccuracy is 3.125 pixels. If, for example, 32×32 textures are used, the inaccuracy would be around 12.5 pixels, which produced enough inaccuracy in some cases.

As can be seen, the best resolution is not absolute but rather relative, depending mainly on the size of elements on screen. However, practically, it is highly unlikely to have objects of, for example, 10 pixels width, and we can safely assume a minimum of 40 pixels width for 800×600 (i.e. 5% of screen width). For this limit, the 3.125 inaccuracy of a 128×128 texture is 7.8125% of the object width, which is acceptable for an object of such size.

3.4 Depth of Field Effect

Unlike other rules for which only the position, orientation, and field of view need to be controlled to satisfy them, the depth of field rule requires controlling camera lens. However, the standard camera model in 3D graphics is a pinhole camera, i.e. the depth of field is infinity, making all objects sharp. To simulate a thin lens camera model (i.e. a camera with a finite depth of field), the depth of field effect has to be manually implemented using blurring filters.

Typically, a minimum and maximum distance are specified, inside which objects appear sharp. Objects outside that range are blurred according to its distance from the range using Gaussian filter. For the technical details of the implementation, the reader is referred to [Riguer et al., 2003].

Now rather than finding the position, orientation, and field of view, the depth of field rule need only return the field parameters, i.e. the near and far distances of objects in focus. For this, the module of this rule (i.e. the module implementing the rule) always return 1 for the rating of satisfaction since the lens parameters are to be decided after finding the other camera parameters. Then, the module will be called again and sent the camera position, orientation, and field of view, and it will be the responsibility of the module to calculate the lens parameters according to the object distance from camera and its approximate size.

3.5 Combining Rules Rating

In most cases it is required to satisfy more than one composition rule. This converts the problem of optimisation to a multi-objective optimisation problem. The difficulty in multi-objective optimisation comes from the requirement of simultaneously optimising two or more conflicting objective functions. In such case, it is not possible to identify a single solution that optimises all objective functions. There is a big literature about multi-objective optimisation, but the problem here is simplified by defining a function representing the total satisfaction of all the rules. This raises the question of how to calculate this function. A typical answer to this question is to use the arithmetic mean of the weighted ratings as follows (see for example [Liu et al., 2010]):

$$Overall\ Rating = \frac{\omega_1 \cdot S_1 + \omega_2 \cdot S_2 + \dots + \omega_n \cdot S_n}{\omega_1 + \omega_2 + \dots + \omega_n} \quad (3.6)$$

where

R_i :rating of the i th rule.

ω_i :weight of i th rule.

The weights are assigned by the user according to the importance of the rule.

One problem with using arithmetic mean is that it is not always suitable when there are many rules applied. For example, if there are 5 rules in a certain configuration (which is not unusual) all of them are fully satisfied except one which is completely unsatisfied, the overall rating will be 80%, which is not what would be expected if one of the important rules is completely unsatisfied. This problem can be alleviated using geometric mean as follows:

$$Overall\ Rating = \sqrt[\omega_1 + \omega_2 + \dots + \omega_n]{S_1^{\omega_1} \cdot S_2^{\omega_2} \cdot \dots \cdot S_n^{\omega_n}} \quad (3.7)$$

As can be seen from the equation, if the rating of one of the rules drops heavily, the overall rating will also drop heavily. Thus, the problem of having unexpected results due to one of the rules being completely unsatisfied yet the overall rating is driven up by other rules is solved.

On the other hand, the problem with geometric mean is that even if some rules are not important in a configuration, the overall rating will be driven down. For this reason, both approaches are

supported in the camera system and the user can select which one to use.

3.6 Conclusion: Limitations in Implemented Rules

This chapter explained the rules selected for implementation. Due to the non-linearity nature of the problem and the aim to produce aesthetically-maximal results, rating functions have been specified for each rule. For the evaluation of the rating functions, two methods were explained, *geometric method* and *rendering method*, of which the rendering method was chosen to get more accurate results. Now that the rules are formulated, the next chapter will discuss the optimisation method used in the camera system.

The beginning of the chapter explained that the motivation behind selecting the mentioned rules is that their impact on the results are clear. But those rules are not enough for real composition and the ability to support more rules is desired. On the other hand, rules sometime need customisation based on the artist's taste. Chapter 5 suggests a methods for solving this problem by presenting a language for composition based on image processing operators.

Chapter 4

Optimisation Framework

Over the past two decades, game developers and researchers developed different methods for solving camera control problems. Each method has its own advantages and disadvantages, making it suitable for certain contexts. In this section, some of those methods which can be used in composition are explained, followed by a discussion of which one should be used in the camera system.

As for the camera control methods which can be used in composition, some of them are:

1. **Vector algebra method:** In this method, the important objects are usually modelled as points, and closed-form vector algebra is used to find the position and orientation of the camera that places one or two objects on certain positions of the screen. This method was first proposed by Blinn (1988) to fix a planet and a moving satellite on certain screen positions. Despite its simplicity, it is the de facto algorithm in many games and simple applications that require fast computations. However, it cannot be used in applications requiring the placement of more than two objects.
2. **Constraints-based method:** In this method, spatial and screen properties are specified as goals to be achieved. Those properties are converted to constraints on the camera position and orientation. Unlike the vector algebra method, this method usually has flexibility in the specification of properties, making it possible to apply properties on many objects. Moreover, the result of this method is usually a volume of feasible camera positions rather than a specific camera position [Christie and Normand, 2005]. The disadvantage of this method is that it lacks precise rating of shots that make it possible to optimise for the best possible shot, which is a requirement in composition.
3. **Optimisation-based method:** In this method, objective functions are constructed according to certain screen properties. The objective functions are used to rate the satisfaction of the screen properties for a certain camera configuration, and an optimisation method is employed to find the best possible camera configuration. For example, Olivier et al. (1999) used genetic algorithms to solve for certain relative and absolute screen properties. The

disadvantage of this method is that it is usually unsuitable for practical applications where the environments are complicated, imposing constraints on the position of the camera, thus requiring the optimisation method to be constrained.

4. **Constrained-optimisation method:** This method is the combination of the constraints and optimisation methods. That is, an optimisation method is run inside a constrained search space. It usually consists of two stages. In the first stage, feasible search spaces are constructed based on properties that restrict the camera placement. In the second stage, a constrained-optimisation method is employed to find the best possible camera position in the feasible search spaces.

Since the camera system supports the imposition of different rules on many objects, and since composition is about the precise placement of screen elements, using optimisation is a necessity. Since the geometry of practical environments impose constraints on the position of the camera, the optimisation problem becomes a constrained one.

4.1 Requirements of the Optimisation Method

There are many different optimisation methods and each has its own advantages and disadvantages. *Hill climbing* is one of the simplest optimisation methods. It starts with an initial solution, checks the neighbouring solutions and then moves to the best. The problem with hill climbing is that it is prone to local minima.

Gradient descent is similar to hill climbing, but depends on the gradient of the function for choosing the direction of movement. Starting with an initial guess, the algorithm moves in the direction of the gradient of the function, based on the fact that a differentiable function decreases fastest in the direction of the gradient. In addition to being a local search method and prone to local minima, it also requires the function to be differentiable.

To alleviate the locality problem in hill climbing, gradient descent, or any local search method, the algorithms can be modified with a random-restart; conducting a series of searches from randomly generated initial states until a solution is found [Russell and Norvig, 2003]. However, this approach is very expensive, making the method impractical.

Since hill climbing is fast but often fails to find solutions while random restart is slow but succeeds in finding solutions, it is reasonable to combine them together, which is what *simulated annealing* does. Unlike hill climbing, simulated annealing chooses a random move rather than the best move. If the new position is better than the current, the method always takes it. However, if it is not, the method accepts it with a probability, which decreases exponentially with how bad the new position is compared to the current position. The probability also decreases with time, making the algorithm less ready to accept worse positions as time progresses.

Another optimisation method is the genetic algorithms. Genetic algorithms mimic the process of evolution in nature. The algorithm starts by selecting a random population (i.e. random positions in the problem space), modelling them as genes. Then a new generation is created by matching random pairs and using the crossover operator, an operator that produces a new gene with information from two parent genes. Furthermore, each element in the new generation is subjected to random mutation with a small independent probability [Russell and Norvig, 2003].

Hill climbing and gradient descent cannot be used in the camera solver because they are prone to locality problem. On the other hand, genetic algorithms and simulated annealing are not suitable because they take a long time until a good solution is reached, especially that the rendering method is used to find the solution. The method used by Liu et al. (2010) and Burelli et al. (2008), *Particle Swarm Optimisation*, avoids these shortcomings, so it is used in the camera system. This method will be discussed in detail in the next section.

4.2 Particle Swarm Optimisation

Particle Swarm Optimisation (hereafter referred to as PSO) is a stochastic optimisation algorithm proposed by Eberhart and Kennedy [Kennedy and Eberhart, 1995] to simulate the social behaviour of swarms in nature. It is based on the hypothesis that individual members of a swarm can benefit from the past experiences of other members [Kennedy and Eberhart, 1995]. More formally, on deciding how to move, each individual depends on three factors:

1. The current direction of movement of the particle; the tendency of a particle to keep moving in the same direction.
2. The best position found so far; the tendency of the particle to move back towards the best *local* position.
3. The best position found by the swarm so far; the tendency of the particle to move towards the best *global* position.

Mathematically, the equation that decides the velocity of a particle is as follows:

$$V_{n+1} = \omega \cdot V_n + C \cdot R_C \cdot (B_L - P_n) + S \cdot R_S \cdot (B_G - P_n) \quad (4.1)$$

where

- P_n :current position of the particle
- V_n :current velocity of the particle
- V_{n+1} :next velocity of the particle
- ω :momentum of the particle
- C, S :cognitive and social factor, respectively
- B_L :local best position
- B_G :global best position
- R_C, R_S :uniformly random values in the range (0, 1)

The factors ω , C and S are the mathematical representation of the three factors described above. The random values R_C and R_S are useful to give some randomisation to the movement of the particles. Note, however, that these random values are actually vector-valued of size N , where N is the dimension of the search space. Also, all the multiplications performed in the velocity equation are component-wise multiplications.

Having the velocity, the next position of a particle is determined by this equation:

$$P_{n+1} = P_n + V_n \quad (4.2)$$

There are many variants for PSO [Pedersen and Chipperfield, 2010], but for the basic equation above, it is obvious that PSO mainly depends on the population size (i.e. number of particles), momentum, and cognitive (C) and social (S) factors.

4.3 Initial Distribution of Particles

PSO starts by randomly distributing the particles in the search space. In the case of hypercubic search spaces, this problem reduces to the simple process of selecting random values for each of the components of the n -dimensional position of a particle (e.g. in this case, randomly selecting values for position, orientation, and field of view). However, the camera system supports polygonal search space, a search space which has the shape of an extruded polygon. In such search space, and generally in non-hypercubic search spaces, random distribution might place a particle outside of constraints. To solve this problem, Hu et al. (2002, 2003) repeatedly produce random positions until all constraints are satisfied. This approach might take a very long time before a position is found. For example, Pulido et al. (2004) demonstrated that in one of their test functions, even the generation of one million random points was insufficient to produce a single solution fully satisfying constraints.

This problem is avoided by finding a random position inside the polygonal base of the polygonal search space, then finding another random value for the height of the particle. To find a random position inside a polygon, a random point on the perimeter of the polygon is found, then another random point between that point and the centre of the polygon is found. This way, a random

position in the search space is found with an even probability over all the search space.

4.4 Search Spaces

Even though the importance of a certain camera configuration is what a camera having this configuration shows, practically the camera position is always restricted to a certain region of space, in addition to restrictions imposed on orientation. For example, in a configuration where two characters are close to a wall, placing the camera beyond the wall renders the characters occluded. In other cases it might be desired to fix the position of the camera even if there is no occlusion. For example, it might be desired to fix the camera on a certain position and get the results through direction and field of view to show the scene from a certain character's perspective. To accomplish this, the approach of using *camera search spaces* is followed. A *camera search space* is a 6-dimensional search space specifying the allowed positions, orientations, and fields of view for a camera. The six dimensions consists of three dimensions for position, two dimensions for direction (as the roll of the camera is always fixed to zero), and one dimension for field of view. There could be many different shapes for camera search spaces, but the following search spaces are found to be sufficient for the examples considered in this thesis:

Point Search Space In this search space, the camera is fixed to a certain position and only the direction and field of view are allowed to change. This is useful in case the camera should be placed in a predetermined position.

Polygonal Search Space This search space has the shape of an extruded polygonal cell. It is defined by a set of vertices, a top, and a bottom. The polygonal shape of the search space makes it flexible to fit in difficult edges in which it is not possible to fit, for example, a search space of a rectangular base.

Multi-Polygonal Search Space Although the *polygonal search space* is quite flexible, there are cases when it becomes problematic. For example, to make an external shot in a dialog between two characters, the camera may be put over the left or right shoulder of one of the characters, but not behind the character, otherwise the other character will be occluded. This raises the need of making a search in multiple disconnected search spaces, necessitating a multi-polygonal search space.

All three search spaces have flexibility in specifying the minimum and maximum values for camera direction angles and field of view.

4.5 Multi-rendering

In chapter 3 it was explained that to process a certain camera configuration, the scene has to be rendered from that certain configuration, then transferred to CPU memory for processing. This operation is expensive, consisting of the following three sub-operations:

- rendering the scene in the GPU;
- transferring the rendered image from the GPU memory to the CPU memory;
- processing the rendered image in the CPU.

The rendering and processing operations cannot be optimised, but the second can be, for transferring large amount of data from GPU memory to CPU memory is faster than transferring multiple small chunks equivalent in size to the big chunk. For example, on an Intel i5-480M CPU and a Mobility Radeon 5470HD with 512MB of memory, the transfer rate of a 128×128 RGBA texture to the CPU memory is around 0.2626 GB/sec while for a 4800×3600 texture the rate is around 0.7768 GB/sec. This suggests rendering the scene from the camera configurations of all particles on a one big offline texture and transferring it to CPU memory in one go.

The multi-rendering method is implemented in the camera system, and rather than using 128×128 texture as suggested in section 3.3, the scene can be rendered from all the 36 particles (the number of particles will be discussion in section 4.7) on a single 768×768 texture. In section 4.10 the performances of the method using single-rendering and multi-rendering are compared.

4.6 Constraining PSO

Having a constrained search space converts the problem of optimisation to a constrained one. However, PSO, in its basic form described above, is an unconstrained optimisation method. There are many methods to make PSO solve constrained problem [Pulido and Coello, 2004, Parsopoulos and Vrahatis, 2002], but the easiest one is the *penalty* method. In this method, the ratings of particles moving to unfeasible positions are reduced so that they have lower values than particles in feasible positions (or they can simply be reduced to zero.) This method is unsuitable here since the search space might be disconnected, meaning that particles might spend long times wandering in unfeasible volumes, which should be avoided. Furthermore, since multi-rendering was used to reduce the GPU-CPU memory transfer, having particles in unfeasible positions means the scene is rendered and the resulting image is transferred to CPU memory even though the camera configuration represented by the particle is unfeasible. For these reasons, a modified version of the penalty method was used, which is termed the *reset-penalty* method.

In the *reset-penalty* method any particle which moves to an unfeasible position is reset (i.e. assigned random position). The argument for this is that after some time, particles having good rating values will be wandering around the best found positions and are less likely to go to unfeasible positions than particles with bad rating values since the latter usually follow other particles which might be in different region of the disconnected search space. So for those particles with bad rating values, it is better to reset them and thus increase the exploration of the space. To support this argument, a test was designed to compare both methods. Four different configurations

		Normal Penalty	Reset-Penalty
Scene 1	Satisfaction	75.3%	86.5%
	Invalid Positions (%)	86.22%	77.22%
Scene 2	Satisfaction	90.9%	95.6%
	Invalid Positions (%)	52.14%	31.15%
Scene 3	Satisfaction	92%	99.5%
	Invalid Positions (%)	91.27%	84.57%
Scene 4	Satisfaction	75.3%	80.7%
	Invalid Positions	53.22%	30.61%

Table 4.1: Comparison between the normal penalty method and the reset-penalty method.

which can hardly reach 100% rating were run. This guarantees that the solver does not stop before executing the maximum number of iterations, which is the same for all test cases. Table 4.1 shows the ratings achieved and the percentages of unfeasible positions during the run time of the test of both methods. As can be seen from the table, the method not only increased the achieved satisfaction, but also decreased the number of particles going to unfeasible positions.

4.7 Tuning PSO Parameters

An important problem in PSO is the selection of the values of momentum (ω), cognitive (C), and social (S) factors and the number of particles. Following the suggestion of Carlisle et al. (2001), the algorithm was configured to use 36 particles¹. A test was then run on a number of different configurations and the value of C and S was changed from 0.2 to 2.6 in steps of 0.2.

As for ω , its value was gradually decreased from some specific value (to be determined by this test) to 0.2 during the run time of the algorithm so that the algorithm balance between global exploration at the beginning and local exploration at the end. To see what is the best starting value for ω , its value was also changed from 0.2 to 1.2 in steps of 0.2.

The results of two of the tests are shown as colour maps in figure 4.1 and figure 4.2. The horizontal scale represents increase in cognitive factor from 0.2 to 2.6 in 0.2 steps, while the vertical scale represents increase in social factor on the same range. Each colour map represents a different value of ω as written on each map. In each colour map, each pixel represents the achieved rating, where red pixels represent better achieved rating and blue pixels represent worse achieved rating. From these figures, the following can be noticed:

1. Decreasing the cognitive and social factors generally improves the achieved rating.
2. Until $\omega = 0.4$, decreasing ω generally improves the achieved rating.
3. For $\omega = 0.4$, which seems to be the best ω for the camera system, the cognitive and social factors achieving the best rating lie in the range from 0.2 to 0.8.

¹ Actually, they suggested 30 particles, but since multi-rendering is used in the system, 36 particles were used rather than 30 to make the width and height of the target texture equal.

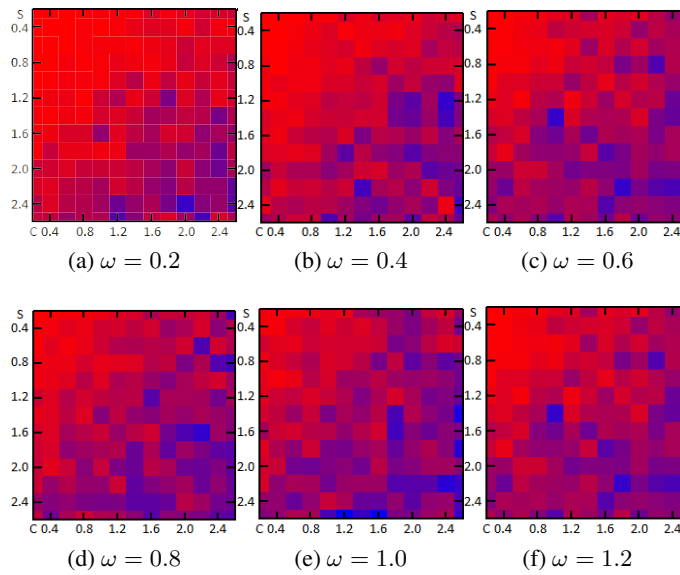


Figure 4.1: Colour maps representing the achieved rating for different values of the PSO parameters for one of the configurations used in the test from section 4.7, where red pixels represent better achieved rating and blue pixels represent worse achieved rating. Each colour map represents a different value of ω , and for each map, the horizontal scale represents increase in cognitive factor from 0.2 to 2.6 in 0.2 steps, while the vertical scale represents increase in social factor on the same range.

These results illustrate that there is no single best set of parameters, and that only good range can be found. So by default the values of 0.5, 0.5, and 0.4 for C , S and ω , respectively, are used in the camera system, but the user of the camera system can manually specify those values if required.

The final question to consider is the number of loops the algorithm should execute before a good satisfaction is reached. Based on trial and error, 100 loops turned out to be enough for all the tests made using the camera system. However, this is not enough to prove that 100 loops are enough, so the user of the camera system can change this number if required.

4.8 The Big Picture of the System

Having discussed all the different components of the camera system, the integration of those parts and the overall picture of the system should be considered. The system consists of two main parts, the optimisation process and the evaluation. The optimisation process is the part where an optimisation algorithm is executed to find the best possible camera position, in this case it is Particle Swarm Optimisation. The evaluation process is the part triggered by the optimisation process to evaluate a certain camera configuration. It consists of three parts: *the renderer*, which is responsible for rendering the ruled objects, *the image processor*, which is responsible for executing image processing functions required by rule modules, and *the rule modules* which connect to the

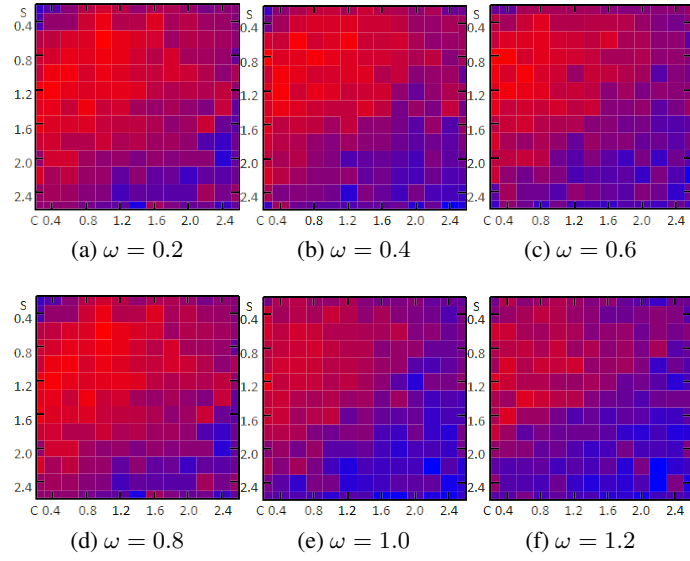


Figure 4.2: Colour maps representing the achieved rating for different values of the PSO parameters for the other of the configurations used in the test from section 4.7, where red pixels represent better achieved rating and blue pixels represent worse achieved rating. Each colour map represents a different value of ω , and for each map, the horizontal scale represents increase in cognitive factor from 0.2 to 2.6 in 0.2 steps, while the vertical scale represents increase in social factor on the same range.

image processor to rate the concerned rules. Figure 4.3 show the full camera system and the connection between the different subcomponents.

4.9 Example: Simple Rules

In this section a practical example of how the rules can be used to produce good shots is presented using the scene shown in figure 4.4a. The goal in this example is to produce a shot of the head in this scene (enclosed by the white frame). Initially, the Rule of Thirds is applied to place the head on one of the power corners. Then, visual balance rule is applied to produce a better shot.

Figure 4.4b shows a top view of the scene in which the green area is the search space used. A large search space is chosen to show the capability of the system in finding satisfactory solutions. Since we want to show a front view of the head, the camera theta and phi angles are restricted to 180 and 0, respectively. We start by trying to apply the Rule of Thirds on the head itself and try to place it on the lower-left *power corner* (described in section 3.1). Having only one rule leads to very few constraints being placed on the solver and by running it repeatedly, the system produced different results, two of them are shown in figures 4.4c and 4.4d. The shot in the first figure, although seemingly good, does not show the environment surrounding the head. The shot in the second figure happened to be good, but we cannot depend on that as a next run of the solver most probably generates a completely different shot due to under constraining.

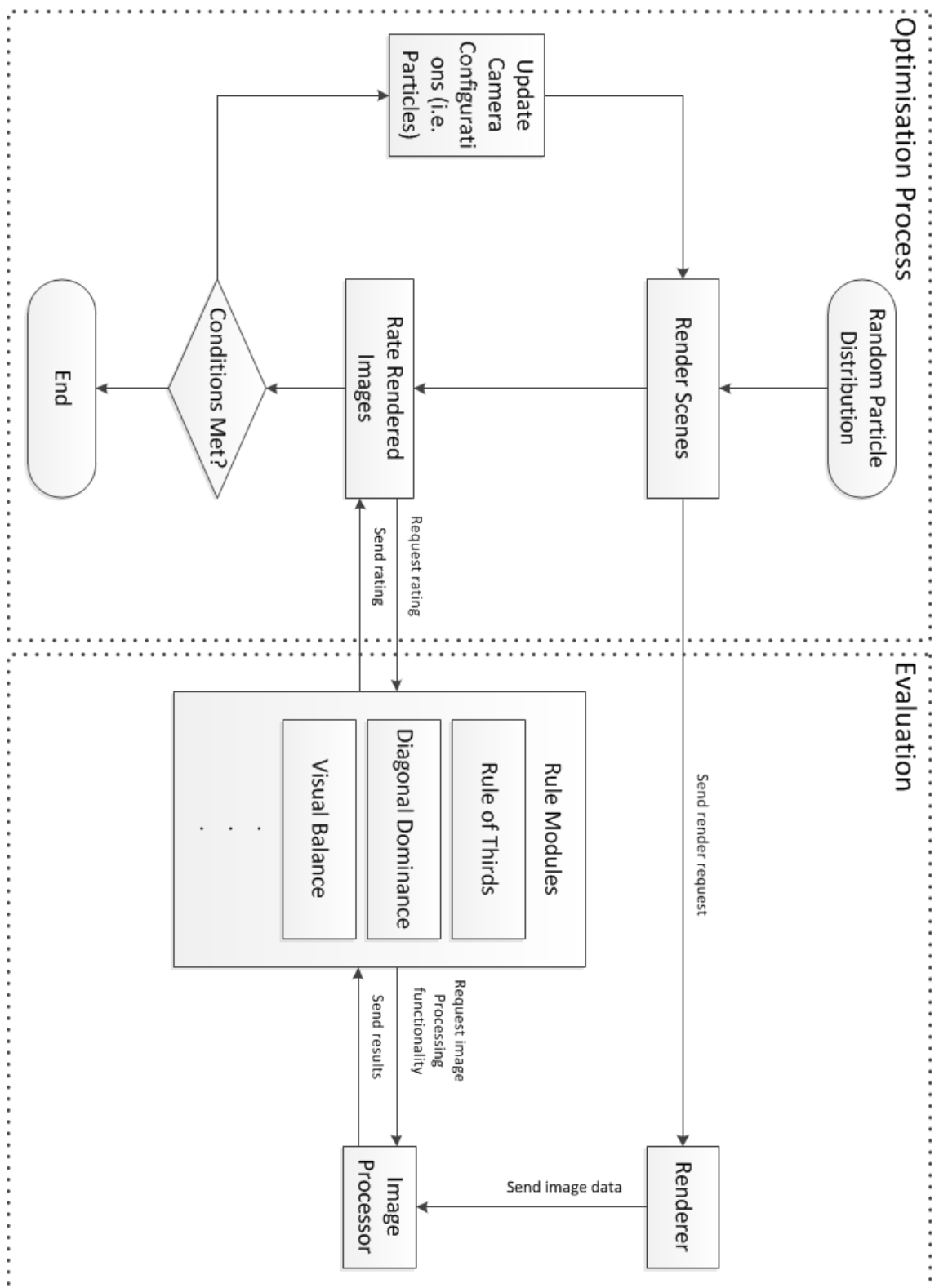


Figure 4.3: Illustrating the subcomponents of the camera system and the communications among them.

To improve the result, another Rule of Thirds was added, but this time on the ceiling rather than the head, so that it is aligned with the upper horizontal line. The result is shown in figure 4.4e. The result is a little better now as the ceiling is aligned with the upper horizontal line, but now the position of the columns make the shot unbalanced. To balance the shot, the visual balance rule is applied on the columns. The result is shown in figure 4.4f.

The final specification of the search space and the applied rules is:

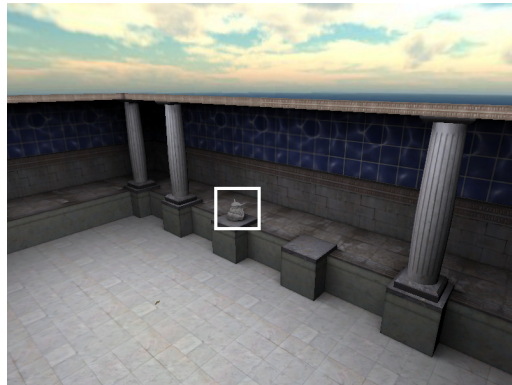
```
SEARCH SPACES
-----
# Define a convex search space.
ConvexSearchSpace
Name: MySpace
# Restrict the theta and phi angles of the camera so that the head is
# front viewed.
Theta: 180,180
Phi: 0, 0
# The upper and lower bounds of the search space.
Top: 200
Bottom: -60
# The vertices of the search space.
Vertex: -180, -400
Vertex: -180, 400
Vertex: 300, 400
Vertex: 300, -400

RULES
-----
# Tell the solver to place the head in the lower left power corner.
RuleOfThirds_Corners
First Object: Head
Enabled Corners: LowerLeft

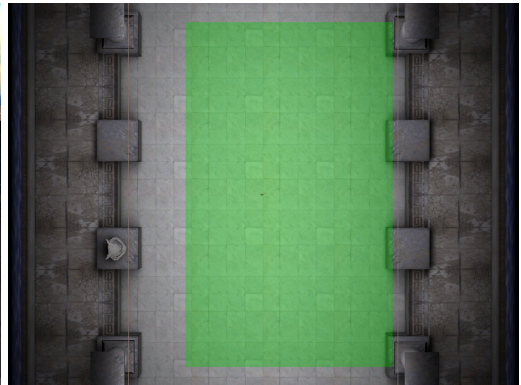
# Tell the solver to align the ceiling with the upper horizontal line.
RuleOfThirds_Alignment
Object: Ceiling
Enabled Lines: Upper

# Balance the columns.
VisualBalance
Object: Columns
Direction: Horizontal
Weight: 3.0

CAMERA SOLVER
-----
# Use the search space defined above.
Search Space: MySpace
```

(a) The scene



(b) Top view



(c) Rule of Thirds on Head



(d) Rule of Thirds on Head - A different shot.



(e) Rule of Thirds on Ceiling



(f) Visual balance on columns

Figure 4.4: Snapshots of Example 1

4.10 Performance Analysis: 1984 Canteen Scene

The second example is about recreation of a scene from Michael Radford's film, 1984. The selected scene is set in a canteen and revolves around 4 important characters in the plot, Smith, Syme, Parsons, and Julia. In the scene, the protagonist, Smith is engaged in conversation with Syme and Parsons at the lunch table, and Julia is watching Smith from across the room. An additional secondary character referred to as OPM (abbreviation of Outer Party Member) participates briefly in one of the conversations. This scene was chosen as its narrative information (character relationships and plot details) is conveyed almost entirely independently of dialog and character actions, instead, relying on cinematography in conjunction with subtle character actions.

The scene has been rendered in 3DSMAX and exported to OGRE, an open source rendering engine. The camera system was used to automatically find camera configurations that show shots similar to those of the original scene. Those camera configurations were then used to generate a video of the rendering of the scene. Table 4.2 lists some of the shots that have been generated and the rules used to generate each shot. For example, the first shot was generated using the **visibility** rule to fully show the gin bottle, and the **framing** rule to place the gin bottle in the top-left part of the shot. In the second shot, the **framing** rule was used to place Smith's head inside a certain frame the coordinates of which are extracted from the original film. The **size** rule is also used to make Smith's head's height approximately equal to his head's height in the original film. Furthermore, the visibility rule is applied three times to show Smith's head and 70% of the screen, and to remove Syme from the shot. For each shot, single-rendering method was used and the test was repeated 10 times to calculate the mean and standard deviation of the achieved rating, processing time, and number of iterations until a solution is found, which are also included in the table in the second column, where the standard deviation is the value in brackets. Finally, the screen coordinates range from (-1, -1) at the bottom-left corner to (1, 1) at the top-right corner.

The system the test was run on had a 2.66 GHz Intel Core 2 Quad Core Q9400 processor and an NVIDIA GeForce 9800 GT video card with 512 MB of video memory. For the rating, scene objects were rendered on an 128×128 offline texture. While it was possible to use smaller texture sizes to reduce processing time this would decrease the accuracy of the results.

The algorithm has been configured to break if the rating reaches 95% or after 100 iterations. As we can see from the table data, the standard deviation of the rating is either zero or negligible, which shows that the results obtained by the system are steady. Another thing to notice is that the processing time varies widely among the different shots because different shots have different rules. Also, the standard deviation of the time is relatively large because, depending on the initial random configurations, the required number of iterations before a solution is reached varies widely. In the fourth configuration specifically, the number of iterations is zero. This is because the position of the camera was fixed to Syme's eyes (to show the view from his perspective) and only the camera pitch and field of view were allowed to change, making it enough for the initial

step to find a satisfactory solution. Finally, the rating of the last shot is relatively low because the used rules cannot be satisfied together.

In section 4.5 it is mentioned that multi-rendering (i.e. rendering the scene from the different configurations at the same time) has some performance gain. To see the performance gain, the same exact test above was repeated with multi-rendering instead of single-rendering, but this time rather than just calculating the total running time of the algorithm, the time spent doing the following was also calculated:

- rendering the scene in the GPU;
- transferring the rendered image from GPU memory to CPU memory;
- processing the rendered image on CPU memory.

The results of those tests are shown in table 4.4. Also, for easier comparison, the same data for single-rendering method is shown in table 4.3. Unlike before, only the averages are shown, and the numbers in parentheses are the percentages spent in each part of the algorithm. Several conclusions can be obtained from these data. First, except for the fourth configuration for which the solution was found in the initial step, the time spent in memory transfer is no more than 15% which means that multi-rendering can only alleviate the time problem, but not remove it. However, the combined time of memory transfer and image processing on CPU forms at least 80% of the total time, which means that if in some way the processing can be done on GPU the total time would be reduced by at least 80%. If this can be fulfilled, and low-poly meshes are used to decrease rendering time, it will be possible to make the rendering method runs in real-time, which is quite promising about the future of this method. This will be briefly discussed in chapter 6.

4.11 Conclusion

In this chapter different solving methods for the composition problem were introduced, and the necessity of optimisation was briefly explained. The chapter also briefly explained different optimisation methods and the shortcomings of some of them that made them inappropriate for the composition problem. Particle Swarm Optimisation was chosen in the system which proved to be very successful in camera control. Constraining PSO to fit in restricted polygonal/multi-polygonal search spaces so that the system can be integrated with other systems was also discussed. Finally, tuning PSO parameters was considered in details to get the best possible results from PSO.

By the end of this chapter, the complete picture of the system has been presented. Chapter 5 addresses the rule modules shown in figure 4.3 and the capability to expand the rule set by implementing modules.

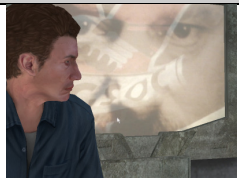
Rules	Rating Time (secs) Iterations	Shot
On bottle and cups		
Visibility(GinBottle, MinVisibility:100%) Framing(GinBottle, MinX:-1.0, MinY:0.0, MaxX:0.0, MaxY:1.0)	98.2% (0.077%) 5.006 (2.061) 63 (26)	
On Smith		
Framing(Smith#Head, MinX:-0.43, MinY:-0.6, MaxX:0.36, MaxY:1) Visibility(Syme, MaxVisibility:0) Size(Smith#Head, MinHeight:1.6) Visibility(Smith#Head, MinVertVisibility:70%) Visibility(Screen, MinVisibility:70%)	95.56% (0.567%) 17.352 (0.515) 100 (0)	
On Syme and OPM		
Size(Syme#Head, MinWidth:0.5, MinHeight:1.2) Size(OPM#Head, MinWidth:0.2, MinHeight:0.4) Framing(Syme#Head, MinX:-0.05, MinY:-0.5, MaxX:0.55, MaxY:0.9) Framing(OPM#Head, MinX:-0.4, MinY:0.0, MaxX:0.0, MaxY:1.0)	99.01% (0.669%) 6.878 (3.032) 54 (24)	
On background characters		
RuleOfThirds_Alignment(<List of Background Characters>, LowerHorizontalLine)	99.43% (0.371%) 0.285 (0.129) 0 (1)	
On Julia		
Framing(Julia#Head, MinX:0.0, MinY:-0.1, MaxX:0.5, MaxY:0.6) Size(Julia#Head, MinWidth:0.25, MinHeight:0.5) DepthOfField(Julia)	99.7% (0.368%) 4.237 (2.137) 48 (24)	
On Smith while listening to screen		
Visibility(Syme, MaxVisibility:0%) Framing(Smith#Head, MinX:-1.0, MinY:0.0, MaxX:0.5) Visibility(Screen, MinVisibility:100%) Size(Screen, MinWidth:2.5, MinHeight:2.5)	72.4% (0.351%) 15.188 (0.11) 100 (0)	

Table 4.2: The list of shots used in the rendering of a scene from Nineteen Eighty-Four. For each shot, we repeated the test 10 times and calculated the average rating the system could achieve and the average time spent in the solving process. The numbers in the brackets are the standard deviation of the results of the 10 tests.

Total time	Rendering time	GPU to CPU transfer time	Image processing time
5.01	0.56 (11.18%)	0.68 (13.57%)	3.68 (73.45%)
17.35	2.72 (15.68%)	2.07 (11.93%)	12.36 (71.24%)
6.88	1.26 (18.31%)	0.76 (11.05%)	4.78 (69.48%)
0.28	0.04 (14.29%)	0.22 (78.57%)	0.03 (10.71%)
4.24	1.40 (33.02%)	0.64 (15.09%)	2.09 (49.29%)
15.19	2.61 (17.18%)	1.97 (12.97%)	10.44 (68.73%)

Table 4.3: The time in seconds spent in the different parts of the solving process for single-rendering.

Total time	Rendering time	GPU to CPU transfer time	Image processing time
3.15	0.36 (11.43%)	0.19 (6.03%)	2.54 (80.63%)
15.94	2.29 (14.37%)	1.20 (7.53%)	12.26 (76.91%)
6.07	0.96 (15.82%)	0.37 (6.1%)	4.65 (76.61%)
0.40	0.08 (20%)	0.26 (65%)	0.05 (12.5%)
4.50	1.34 (29.78%)	0.34 (7.56%)	2.69 (59.78%)
13.85	2.13 (15.38%)	1.10 (7.94%)	10.45 (75.45%)

Table 4.4: The time in seconds spent in the different parts of the solving process for multi-rendering.

Chapter 5

C for Camera Composition

In chapter 3 it was mentioned that a limitation in the camera system lies in that the number of implemented composition rules is limited. In this chapter, a solution for this problem is proposed by presenting a language for composition. The goal of the language is to:

Enable the user to implement composition rules by rating images using compositionally useful functionalities.

In its current version, the language is not supposed to allow full implementation of all the rules in composition, but it is a start into this field. As an example, a rule will be implemented using the language to demonstrate its capability in implementing rules easily. The rule to be implemented is one of the rules already built in the system. This enables comparing the performance of both cases.

The language is called “*C for Camera Composition (Ccc)*”, mimicking the name of the Intel Shader language “C for Graphics” since it also has C-based syntax. Finally, the language is an interpreted one, but the code is interpreted before execution and converted to bytecode for faster execution.

5.1 Requirements

Depending on the context, each programming language has its own requirements. To know the requirements of Ccc, the goal of using the language (i.e. building composition rules) should first be detailed:

1. The language should enable the user to easily process images in a compositionally fruitful way.
2. The language should make it easy to build rating functions with a rich set of build-in functions.

3. The language should be integrated with the camera system to allow the user to reference objects in scene and communicate the results of processing and rating back to the camera system.

To implement these goals, the language should fulfil the following requirements:

- **Variable Types:** The variable types that need to be supported in this language are different than ordinary programming languages. In addition to integer, float, etc., the language should also support variables like point, line, etc., which are used in many image processing functionalities, e.g. finding the prominent line of a screen element returns a variable of type **line**.
- **Built-in Functions:** The language should support built-in functions that are helpful in dealing with graphic, photographic, and image contents to make the task of implementing rules easier for the user.
- **Communication With Camera System:** To define usable rules, the user should be able to reference elements in the virtual environment, i.e. elements that the rule should be applied to. Moreover, many rules usually require additional parameters like weight, tolerance, etc. The language should also allow passing of such parameters.
- **Direct Image Access:** Although some image processing functions are already built in the language, they are limited, so the user should be able to support additional functions. Without direct access to the pixels of the image containing the scene rendering, it is impossible to implement this.
- **Standard Requirements:** Like all programming languages, it is necessary to have arithmetic operations, comparison operators, and similar functionalities. Moreover, since many rules have different cases for different parameters and sometimes require iterations, the language should also support conditional and loop statements. For example, in the Rule of Thirds, the user should be able to specify which power corner(s) the element under consideration should be placed at, for which an if statement is required. Additionally, it is also desirable to allow the user to define functions to better structure the code.
- **Interpreter:** An interpreter should be implemented and integrated within the camera system to allow the use of the language within the camera system.

The following sections will be devoted to describing how each of these requirements is implemented in the language. Then, another section will be devoted to briefly discussing the implementation of the parser.

5.2 Variable Types

The language currently requires the following variable types:

- *Number*: It is a necessity in any programming language to support number variable types. Many programming languages support different types of numbers like integer, unsigned integer, long, float, double, etc. The differences are caused by the sign bit (e.g. integer vs. unsigned integer) and the size of the variable (e.g. integer vs. long). Since these differences are not important in Ccc, only two number variable types are supported: **int**, which is a 32-bit integral value, and **real**, which is a 32-bit IEEE 754 floating point number.
- *Point*: It is vitally important to have point variable types to reference objects on screen. For example, to rate the satisfaction of the Rule of Thirds, it is required to find the distance between the centroid of the important screen element and the nearest power corner. This variable type is used to represent 2D points.
- *Line*: Besides numbers and points, lines are also required because they represent an important element in the graphic contents of an image. For example, in the diagonal dominance rule, it is required to find the prominent line of an object so that it is compared to the screen diagonals. The line variable type is used in such cases.
- *Rectangle*: In addition to lines, it is also necessary to support rectangles for functions that use it like the **boundary** function which finds the rectangle bounding a certain screen element.
- *Variable set*: As the name suggests, this is a set of many variables combined under a single name. An example use of a variable set is the **left_boundary** function which returns the positions of the left-most pixels of a screen element.
- *Screen Element*: An important part of the language is the communication with the camera system. Every rule is applied to elements on screen, so there must be a way of referencing those element in the language. This variable is used for this task.

5.3 Built-in Functions

The language supports two types of built-in functions: image processing functions and general-use functions.

5.3.1 Image Processing Functions

As was mentioned in the beginning of this chapter, a key feature in a language used to build rules in an image processing-based composition system is to support image processing functionality, first

to make the task of writing code in the language easier for the user, and second to avoid having to write expensive code in an interpreted language as this will heavily slow down the processing. In the current version of the language, the following image processing functions have been supported:

- **centroid**: calculates and returns the centroid of a screen element;
- **width**: finds the maximum sectional width of a screen element;
- **height**: finds the maximum sectional height of a screen element;
- **average_width**: finds the average of sectional widths of a screen element;
- **average_height**: finds the average of sectional heights of a screen element;
- **prominent_line**: finds the prominent line of a screen element;
- **boundary**: finds a rectangle bounding a screen element;
- **upper_boundary**: finds the set of pixels representing the upper boundary of a screen element and return them as a set of pixels;
- **lower_boundary**: finds the set of pixels representing the lower boundary of a screen element and return them as a set of pixels;
- **left_boundary**: finds the set of pixels representing the left boundary of a screen element and return them as a set of pixels;
- **right_boundary**: finds the set of pixels representing the right boundary of a screen element and return them as a set of pixels.

5.3.2 General-use Functions

Those are functions that generally help making the task of building rules easier. Currently, the following functions are supported:

- **average**: Finds the average of a set of numbers;
- **distance**: Finds the distance between two points;
- **gaussian**: Finds the Gaussian satisfaction of a certain parameter. Gaussian satisfaction is the standard satisfaction equation used in the camera solver (see section 3.2.4);

$$FR = e^{(\frac{\Delta - \Delta_t}{\Delta_{do} - \Delta_t})^2} \quad (5.1)$$

- **manhattan**: Finds Manhattan distance between two points;

- **nearest_point**: From a set of given points, find the one nearest a given point.
- **power**: Find the power of a number raised to another number.
- **standard_deviation**: Finds the standard deviation of a set of numbers.

5.4 External Communication

A requirement of the language is to be able to reference elements from the virtual environment. In section 5.2, it was mentioned that the **screen element** variable type is used to reference screen elements (i.e. from the virtual environment). But the question of setting the value of screen elements, along with other required parameters, remains open.

This is implemented in the language using a special keyword, the **input** keyword. The **input** keyword indicates that the value of a variable should have its initial value defined by the camera solver. The camera solver on the other hand allows setting the value of those variables in the definition of user rules, which are rules whose rating are calculated using a Ccc code.

For example, a typical Rule of Thirds definition may contain the following **input** variables:

```
input screen_element firstElement;
input screen_element secondElement;
```

These two statements tell the interpreter that these two variables should have their values pre-specified before the execution of the code. These values are set in the *user rule module* (see section 5.8).

5.5 Direct Image Access

To allow the user to implement custom image processing functions, apart from the built-in ones, the language should provide some way to directly access the pixels of the image containing the rendering of the scene from a certain camera position. There are two methods to achieve this:

- enable access of the image pixels from the language with an interpreted code;
- enable access of the image pixels from the language with a compiled code.

The problem with the first method is that it is expensive as the language code is converted to bytecode rather than machine code. However, since the second method is too complex to implement and require building a compiler which is beyond the scope of this thesis, the first method is chosen.

A special function, **image_pixel(x, y)**, has been implemented in the language to allow accessing the pixels of the image. The user can then iterate through the pixels of the image and use the supported functionalities of the language to process the image.

Arithmetic Operators			
$a + b$	Add 'a' to 'b'	$a - b$	Subtract 'b' from 'a'
$a * b$	Multiply 'a' and 'b'	a / b	Divide 'a' by 'b'
$a++$	Increment 'a' by one.	$a--$	Decrement 'a' by one.
$-a$	Negative of 'a'		
Comparison Operators			
$a == b$	'a' equals 'b'	$a != b$	'a' does not equal 'b'
$a < b$	'a' smaller than 'b'	$a <= b$	'a' smaller than or equal to 'b'
$a > b$	'a' greater than 'b'	$a >= b$	'a' greater than or equal to 'b'
Bit-wise Operators			
$a \& b$	Bit-wise AND between 'a' and 'b'	$a b$	Bit-wise OR between 'a' and 'b'
$a \wedge b$	Bit-wise XOR between 'a' and 'b'		

Table 5.1: The operators supported in Ccc

5.6 Standard Requirements

In addition to the requirements mentioned above, some of the standard requirements of every programming language are also implemented. Those include arithmetic operators, comparison operators, bit-wise operators, conditional statements, and loop statements. Table 5.1 list the supported operators.

Also, the *if* conditional statement and the *for* loop statement are also supported and their syntax is the same as their counterparts in C/C++ language. Finally, since the code for implementing a rule can easily get complicated, supporting functions in the language improve the usability and readability of the code. Function can be defined and called in exactly the same way as in C/C++ language.

5.7 Interpreter

The interpreter of the language was built with a compiler generator. A compiler generator is a tool that generates source code for a parser, interpreter, or compiler using some formal description of the language, e.g. BNF (Backus-Naur Form). Specifically, Lex & Yacc were used to build it. Lex & Yacc are actually a two-tool compiler generator, the first, Lex, being the lexical analyser and the second, Yacc, being the parser. Lex accepts a table of regular expressions that specifies the tokens to be extracted from a source code. It then generates source code that translates source code into a list of tokens.

Yacc, on the other hand, accepts a list of grammar rules specified in a form similar to BNF. It then generates source code that accepts tokens (which are usually generated by Lex, though any other tool can be used) and match them with grammar rules. The action performed when a grammar rule is matched is to be defined by the user. For the grammar of Ccc, bytecode is directly generated according to the parsed rule. For example, the simple statement `"int i = a*b;"`

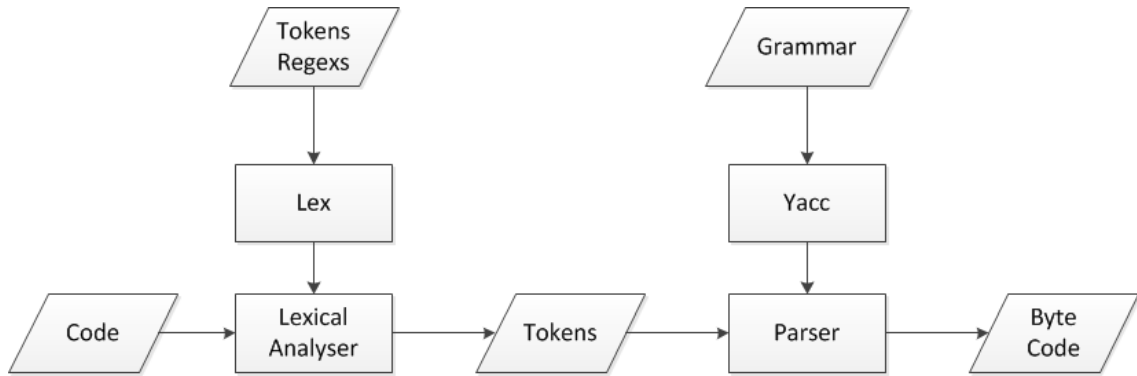


Figure 5.1: Having regular expressions specifying tokens and grammar rules, Lex & Yacc can be used to generate an interpreter that convert source code to byte code.

generates the following byte code

```

define int, i
push i
push a
push b
multiply
assign

```

The regular expressions of Lex and the grammar rules of Yacc are left to appendix A, but figure 5.1 illustrates the different parts of the interpreter.

5.8 User Rule Module

To be able to integrate the language in the camera system, there should be a customisable module which accepts Ccc code. This module is called *user module*. Like other modules, the user rule module is triggered by the optimisation process for rating. However, unlike other modules which are hard coded, the user rule module execute Ccc byte code to rate the satisfaction of the rule. Moreover, the parameters specified by the Ccc code (i.e. variables defined via the **input** keyword) need to be specified before the execution of the rule. With the integration of the Ccc language in the system, the big picture of the system shown in figure 4.3 changes to the one shown in figure 5.2.

5.9 Practical Example: Rule of Thirds

This section presents a simple example of using the language. The example code will implement the power-corners part of the Rule of Thirds. Repeating the definition from chapter 3, the *Rule of Thirds* proposes that “a useful starting point for any compositional grouping is to place the main subject of interest on any one of the four intersections [called the *power corners*] made by two equally spaced horizontal and vertical lines” [Ward, 2003, p. 124]. To simplify the rule, let’s first consider applying the rule to only one object. The problem then comes down to identifying the

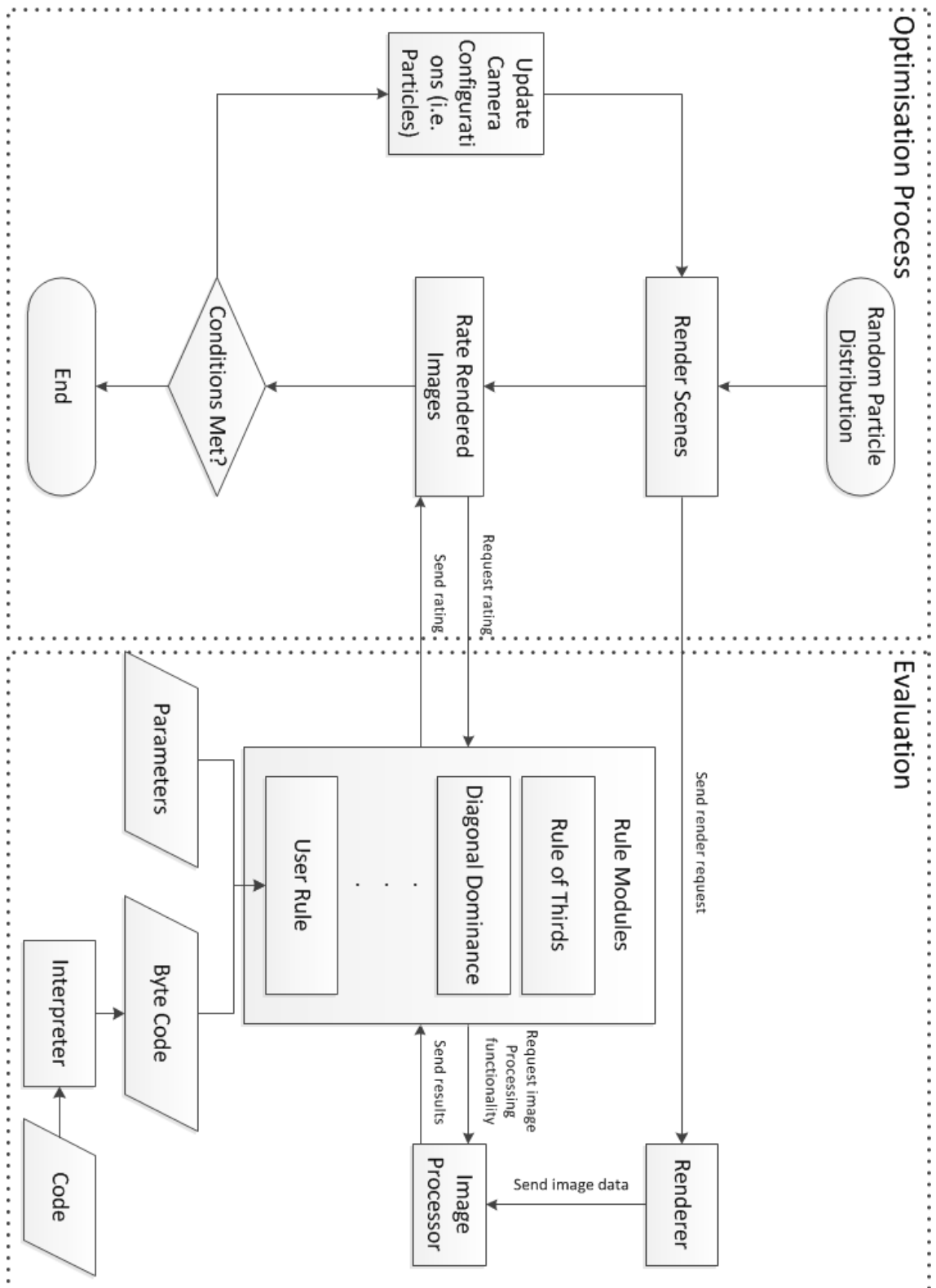


Figure 5.2: The complete camera system.

power corner the object should be at and rate the satisfaction of the rule according to its distance from the object's centroid. For this, we simply find the nearest power corner.

```
// The input keyword is like the "uniform" keyword in Shader languages, it expects
// initial value from the calling context. In this case, it is the names of the
// objects the rule is to be applied to (i.e. Smith and Syme).
input screen_element firstObject;

// Define the power corners, and put them in a set.
point g_cornerUR = { 0.33333, 0.33333};
point g_cornerUL = {-0.33333, 0.33333};
point g_cornerLL = {-0.33333, -0.33333};
point g_cornerLR = { 0.33333, -0.33333};
set g_powerCorners = {g_cornerUR, g_cornerUL, g_cornerLL, g_cornerLR};

// Find the centroid of the object.
point elpos = centroid(firstObject);

// Find the nearest power corners to the centroid of the object.
point p1 = nearest_point(elpos, g_powerCorners);

// Find the distances between the centroid and the nearest power corners.
real eldist = distance(elpos, p1);

// Finally, find and return the rating of the rule using 'gaussian' function.
return gaussian(eldist, 0.0, 0.33333/10, 0.33333);
```

In the last line of this code, value 0.33333 is used for the drop-off value, which is half the distance between two power corners. That is, if the element lies in the middle of two power corners, then the rating should drop off heavily. The value for tolerance is 0.33333/10, which is one-tenth of the drop-off value.

The Rule of Thirds might be applied to more than one object. Theoretically, it can be applied to four objects, but practically, the most common cases are either one object or two object, so we leave the other cases. To enable the rule to be applied to more than one object, we need to repeat the same code for a second object and multiply the results from both objects.

```
input screen_element firstObject;
input screen_element secondObject;

point g_cornerUR = { 0.33333, 0.33333};
point g_cornerUL = {-0.33333, 0.33333};
point g_cornerLL = {-0.33333, -0.33333};
point g_cornerLR = { 0.33333, -0.33333};
set g_powerCorners = {g_cornerUR, g_cornerUL, g_cornerLL, g_cornerLR};

point elpos = centroid(firstObject);
point e2pos = centroid(secondObject);

point p1 = nearest_point(elpos, g_powerCorners);
point p2 = nearest_point(e2pos, g_powerCorners);

real eldist = distance(elpos, p1);
```

Total time	Rendering time	GPU to CPU transfer time	Image processing time
Using built-in rule of thirds			
11.73	3.96 (33.76%)	0.56 (4.77%)	7.06 (60.19%)
Using Ccc-based rule of thirds			
12.14	3.91 (32.21%)	0.61 (5.02%)	7.47 (61.53%)

Table 5.2: The time in seconds spent in the different parts of the solving process for multi-rendering.

```
real e2dist = distance(e2pos, p2);

return gaussian(e1dist, 0.0, 0.33333/10, 0.33333)*gaussian(e2dist, 0.0, 0.33333/10,
0.33333);
```

Since Ccc is an interpreted language, it is important to study the performance overhead of using it. To see this, test on two configurations was run, one using the built-in rule of thirds, and the other using the rule of thirds declared above using Ccc. Like before, the test was repeated 10 times in each case. The data in table 5.2 shows the average time spent in the run-time course of each configuration.

Although the rendering and transfer time should be the same (apart from small variations among different runs) in both cases, but I included them to compare the overhead in processing code due to byte code execution relative to the time of other parts. As can be seen, the increase in the processing time is 0.41 of a second. This includes the execution of 137 lines of byte code $36 \times 100 = 3600$ times, which is fast relative to the total processing time.

5.10 Conclusion

In this chapter a language for composition in camera control was proposed. The chapter started with the motivation of the language, followed by describing the requirements of the language like variables and built-in functions. The chapter also described how the language integrates with the camera system through the **input** keyword. Though the built-in functions include many functions for image processing, the user can extend those functions by directly accessing the image to implement more image processing functionalities. The chapter also described the interpreter of the language and how it was implemented. The main short coming of the language is that it cannot be implemented on the GPU, restricting its use for off-line mode only.

Chapter 6

Conclusion

This chapter summarises the contents of the thesis and discusses the limitations of the presented camera control system. It concludes with a section about the possible future work.

6.1 Summary

The thesis presented a camera control system that considered compositions in photography and cinematography. A number of previous researches already considered the problem of composition in camera control, but had one or more of the following three main shortcomings:

1. The lack of support of advanced composition rules (e.g. rule of thirds) in camera control systems.
2. The approximation of complicated 3D models with approximate geometric shapes, making the solution inaccurate.
3. The limited applicability of many methods to few objects and few rules.

The camera system presented in this thesis is an attempt to solve these three main shortcomings. The first shortcoming, which is the main topic of the thesis, was addressed by exploring some of the most important rules in the literature of photography and cinematography and implementing four of them in the camera system: rule of thirds, diagonal dominance, visual balance, and depth of field. The second shortcoming was addressed using the rendering method, as opposed to the geometric method (see section 3.2.1) which is used in many previous camera systems. Finally, the last shortcoming was addressed by using an optimisation method, allowing the application of many rules on many objects.

As an optimisation-based camera system, rating functions were formulated for each of the implemented rules. A set of factors were specified for each rule to determine its rating, which is the geometric mean of the ratings of all factors. The rating function of any factor is calculated based on the rendering method, in which the silhouette of scene objects are rendered on an off-line texture, which is then transferred to CPU memory for processing. For example, to evaluate

the satisfaction of the diagonal dominance rule, the prominent line is extracted from the pixels of the object using perpendicular linear regression [Weisstein, 2010]. Then, having the prominent line, the rule is rated according to the angle of the prominent line and its distance relative to the diagonal of the image.

The camera system uses Particle Swarm Optimisation (PSO) to solve for the best possible camera configuration. PSO was chosen because it is known to work well in higher dimensional spaces [Burelli et al., 2008, Liu et al., 2010]. Furthermore, it also avoids local minima, which is a requirement for the camera control problem. PSO depends on four parameters: the number of particles used, momentum (ω), cognitive factor (C), and social factor (S). Following the suggestion of Carlisle et al. (2001), 36 particles were used¹. The values of the other three parameters were assigned to 0.5, 0.5, and 0.4 for C , S and ω , respectively. These values were chosen based on the meta-optimisation process described in section 4.7.

Finally, one generic problem with many camera systems is the limited number of implemented rules. To solve this problem, a language for composition was presented to allow the user to build his/her rules and have the camera system solve for them. The language is based on image processing operations, which can be used to implement rating functions. For example, to implement the rule of thirds, the user can use the **centroid** function to find the centroid of an object, then use the **gaussian** function to find the rating of the rule according to the distance between the centroid of the object and the closest power corner. The language is rich of such functions which help the user in implementing composition rules. Additionally, the user can have direct access on the image pixels, allowing the implementation of more image processing functionalities in addition to the built-in ones.

6.2 Limitations

There are some limitations in the camera system:

1. **Non-Real-Time Performance:** The most important limitation in the camera system is the performance, since the system does not work in real-time. The reason for that is the use of rendering method which is expensive since it requires objects to be rendered on an offline texture, and the latter needs to be transferred to CPU memory for processing.
2. **No Occlusion Detection:** The camera system does not take into considerations the fact that the target object(s) might be occluded by other objects in the scene, which causes problems in the produced shot. The main reason for not considering occlusion is that it requires all potential occluders to be rendered, adding more cost to the solving process which is already expensive.

¹Actually, they suggested 30 particles, but since multi-rendering is used in the system, 36 particles were used rather than 30 to make the width and height of the target texture equal.

3. **Limited Depth of Field Rule:** The current implementation of the depth of field rule is only a final step to adjust the lens of the camera to put the main subject in focus. This might be a problem in case the solution found happens to put other than the main subject at the same distance from the camera, making it in focus as well.

6.3 Future Work

The future work can mainly focus on solving the limitations of the system. The most important limitation is the *non-real-time performance*. As can be seen from table 4.3 and 4.4, the major part of the processing time is spent in GPU to CPU memory transfer and image processing. This suggests that if the image processing is done in GPU rather than CPU and low-poly 3D models are used in the solving process, real-time performance might be achieved.

There has already been considerable work on doing computation on GPU to exploit the multi-processing environment of GPU [Kirk and Hwu, 2010, Hwu, 2011]. For example, sorting can be implemented on GPU via a modified merge sort that can exploit the multi-processing environment of the GPU [Pharr and Fernando, 2005]. Similarly, algorithms for image processing have been implemented on GPU [Hwu, 2011]. If the image processing operations used in the camera solver are implemented on GPU, it will be possible to remove the overhead of transferring from GPU to CPU memory and the processing in CPU.

As an example, the framing rule is rated based on the frame surrounding an object. The latter can be easily found in the GPU using multi-pass rendering, which produces the coordinates of the frame in the last pass as follows. First, the scene is rendering normally. Then, the first pass uses a fragment program that for each 4×4 block generates a 2×2 block containing the coordinates of the frame surrounding the object pixels in the original 4×4 block. The successive passes then find the coordinates of the rectangle containing all the 4 rectangles represented by the 4 2×2 blocks contained in each 4×4 . The coordinates of the rectangle is then stored in a 2×2 in the next pass. This process is repeated until the final pass produces a 2×2 image containing the coordinates of the frame surrounding the object under consideration. Following a similar approach, all image processing operations can be implemented on GPU.

The future work can also improve the depth of field rule. If it happens that other objects have equal or nearly equal distance to camera as that of the object targeted by the rule, those objects will also be in focus. To solve this problem, the rule should be integrated in the optimisation process so that the obtained camera configuration puts the target object in different distance to the camera than other important objects.

The occlusion problem also needs to be considered in the future work. The main problem with occlusion is that all potential occluders need to be rendered to assess visibility. This, however, heavily increases the cost of processing. A possible idea to alleviate this problem is to assess occlusion through the use of ray-casting method, which although approximate, is quite fast to run

in real-time.

There are four types of compositional controls: graphic controls, photographic controls, colour controls and total control. As a first attempt into composition in camera control, the camera system considered conventions from graphic and photographic controls. A possible future work is to implement rules from the other compositional controls.

Finally, Ccc can be improved to support more image processing operations. Furthermore, more rules from the literature of composition can be explored and implemented using the language to prove its usefulness.

Appendix A

Ccc Interpreter

A.1 Lexical Analyser

The first step in interpreting Ccc code is to divide the code into tokens that the parser uses to generate final code (i.e. bytecode in this case.) The lexical analyser was built using Lex, a lexical analyser generator. Lex accepts a table of regular expressions and generates C source code that translates code into a list of tokens. The input file consists of two sections separated by a line containing two consecutive perecentage signs (%%). The first section is the *definitions* section which gives names to some regular expressions which are used in the second section, the *rules* section, to encapsulate the token in a C language value. For example, **INTEGER** in the first section is defined by the regular expression “[0-9]+” which matches any integral number in the source file. This definition is used in the second section in a rule that returns a C constant (having the name **INTEGER**, which should not be confused with the definition name) that can be used to signal to the parser the appearance of a number in the source file. The Lex input file is shown below:

```

INTEGER      [0-9]+
REAL         [0-9]+\.[0-9]*|-?[0-9]+\.[0-9]+
NAME         [a-zA-Z][a-zA-Z0-9_]*
COMMENT      \\|\/.*\n

%%

{COMMENT}    { /* skip comments */ }
\n           { /* skip new lines */ }
[ \t]+       { /* skip blanks */ }
print        { return PRINT; }
return       { return RETURN; }
for          { return FOR; }
if           { return IF; }
else         { return ELSE; }
input        { return MYINPUT; }
\+\+        { return INCREMENT; }
\-\-        { return DECREMENT; }
\+          { return ADD; }
```

```

\ -      { return SUBTRACT; }
\ *      { return MULTIPLY; }
\ /      { return DIVIDE; }
\ ^ \ ^  { return POWER; }
\ |      { return OR; }
\ &      { return AND; }
\ ^      { return XOR; }
==       { return EQUAL; }
!=       { return NOTEQUAL; }
\ >      { return GREATER; }
\ >=     { return GREATEREQUAL; }
\ <      { return SMALLER; }
\ <=     { return SMALLEREQUAL; }
=        { return ASSIGN; }
\[ \] \. { return MEMBEREXTRACTION; }
\.       { return MEMBER; }
{INTEGER} {
    sscanf(yytext, "%d", &yylval.integer);
    return INTEGER;
}
{REAL}    {
    sscanf(yytext, "%f", &yylval.real);
    return REAL;
}
{NAME}    {
    strcpy(yylval.string, yytext);
    return NAME;
}
.         {
    // Pass all other characters as they are.
    return yytext[0];
}

```

A.2 Grammar

Having the tokens produced by the lexical analyser, the parser combines them to form useful statements according to some rules. The parser is generated by Yacc which accepts an input file containing the grammar of the parser and generates C source code implementing the parser. The input file consists of two sections. The first section contains the list of tokens which are used in the grammar rules in the second section and the priorities of the operators used in the code. The second sections contains the definitions of non-terminal symbols, which are the symbols generated by a combination of non-terminal symbols and tokens (which are also called terminal symbols) according to some grammatical rule. The Yacc input file is shown below:

```

%start statements

%token INTEGER
%token REAL
%token NAME
%token PRINT

```

```

%token RETURN
%token IF
%token FOR
%token ELSE
%token MYINPUT
%token INCREMENT
%token DECREMENT
%token EQUAL
%token NOTEQUAL
%token ASSIGN
%token GREATER
%token GREATEREQUAL
%token SMALLER
%token SMALLEREQUAL
%token MEMBER

%left MEMBEREXTRACTION
%left EQUAL
%left NOTEQUAL
%left GREATER
%left GREATEREQUAL
%left SMALLER
%left SMALLEREQUAL
%left ADD SUBTRACT
%left MULTIPLY DIVIDE
%right POWER
%left INCREMENT
%left DECREMENT
%left AND
%left OR
%left XOR
%left UMINUS

%%

statements      : /* empty */
                | statements statement
                | statements FOR '(' statement expression ';' expression ')'
                  '{' statements '}'
                | statements IF '(' expression ')' '{' statements '}'
                | statements IF '(' expression ')' '{' statements '}'
                  ELSE '{' statements '}'
                | statements function_definition '{' statements '}'
                ;

statement       : NAME variable_definition ';'
                | MYINPUT NAME variable_definition ';'
                | NAME ASSIGN expression ';'
                | RETURN expression ';'
                | PRINT '(' expression ')' ';'
                ;

expression      : expression ADD expression
                | expression SUBTRACT expression

```

```

| expression MULTIPLY expression
| expression DIVIDE expression
| expression POWER expression
| SUBTRACT expression %prec UMINUS
| expression AND expression
| expression OR expression
| expression XOR expression
| INCREMENT expression
| DECREMENT expression
| expression INCREMENT
| expression DECREMENT
| expression EQUAL expression
| expression NOTEQUAL expression
| expression GREATER expression
| expression GREATEREQUAL expression
| expression SMALLER expression
| expression SMALLEREQUAL expression
| NAME
| NAME MEMBER NAME
| expression MEMBEREXTRACTION NAME
| '(' expression ')'
| NAME '[' expression ']'
| NAME function_args
| variable_set
| REAL
| INTEGER
;

function_definition : NAME NAME type_name_list ;

type_name_list      : '(' ')'
| '(' type_name_list_body ')'
;

type_name_list_body : NAME NAME
| type_name_list_body ',' NAME NAME
;

variable_definition : NAME
| NAME ASSIGN expression
| variable_definition ',' NAME
| variable_definition ',' NAME ASSIGN expression
;

function_args        : '(' variable_set_body ')'
;

variable_set         : '{' variable_set_body '}'
;

variable_set_body    : /* allow empty variable sets */
| expression
| variable_set_body ',' expression
;

```

Appendix B

Vertex and Fragment Programs

As mentioned in section 3.2.2, a vertex program is used to bring the pixels resulting from rendering towards the centre of the screen such that the horizontal and vertical distances to the centre of the screen are half their original values. This is the code of the vertex program:

```
void flatVertexShader
(
    // Vertex position in model space
    float4 position      : POSITION,
    // Texture UV set 0
    float2 texCoord0     : TEXCOORD0,
    // Transformed vertex position
    out float4 outPosition : POSITION,
    // UV0
    out float2 uv0        : TEXCOORD0,
    // World-View-Projection Matrix
    uniform float4x4 worldViewProj,
    // Centralization factor (usually 0.5)
    uniform float scaleFactor
)
{
    // Calculate output position
    outPosition = mul(worldViewProj, position);

    // Move the pixel toward the center of the screen
    // such that the horizontal and vertical distance
    // between it and the center is half the original
    // distance.
    outPosition.x = outPosition.x * scaleFactor;
    outPosition.y = outPosition.y * scaleFactor;
}
```

Also, section 3.2.3 mentioned that silhouette rendering is used to make the process of rating images easier. This is the code of the fragment program:

```
void flatFragmentShader
(
    float2 uv0          : TEXCOORD0,
    out float4 color     : COLOR,
    out float depth      : DEPTH,
```



```
    // The colour to render the object width.  
    uniform float4 renderColor,  
    // A distinct index to distinguish the  
    // object being rendered.  
    uniform float4 objectIndex  
)  
{  
    color = renderColor;  
    depth  = objectIndex.x;  
}
```

Bibliography

- [Arijon, 1976] Arijon, D. (1976). *Grammar of The Film Language*. Focal Press.
- [Banerjee and Evans, 2004] Banerjee, S. and Evans, B. L. (2004). Unsupervised automation of photographic composition rules in digital still cameras. In *In Proceedings of SPIE Conference on Sensors, Color, Cameras, and Systems for Digital Photography*, volume 5301, pages 364–373.
- [Bares, 2006] Bares, W. (2006). A photographic composition assistant for intelligent virtual 3d camera systems. In *Smart Graphics*, volume 4073 of *Lecture Notes in Computer Science*, chapter 16, pages 172–183–183. Springer Berlin / Heidelberg, Berlin, Heidelberg.
- [Bares et al., 2000a] Bares, W., McDermott, S., Boudreaux, C., and Thainimit, S. (2000a). Virtual 3D Camera Composition from Frame Constraints. In *Proceedings of the eighth ACM international conference on Multimedia*, pages 177–186. ACM.
- [Bares and Lester, 1998] Bares, W. H. and Lester, J. C. (1998). Intelligent multi-shot visualization interfaces for dynamic 3d worlds. In *Proceedings of the 4th International Conference on Intelligent User Interfaces*, pages 119–126. ACM.
- [Bares et al., 2000b] Bares, W. H., Thainimit, S., and McDermott, S. (2000b). A model for constraint-based camera planning. In *Smart Graphics AAAI Spring Symposium*, pages 84–91.
- [Blinn, 1988] Blinn, J. (1988). Where am i? what am i looking at? *IEEE Computer Graphics and Applications*, 8(4):76–81.
- [Burelli et al., 2008] Burelli, P., Gaspero, L., Ermetici, A., and Ranon, R. (2008). Virtual camera composition with particle swarm optimization. In *Proceedings of the 9th international symposium on Smart Graphics*, SG '08, pages 130–141, Berlin, Heidelberg. Springer-Verlag.
- [Byers et al., 2004] Byers, Z., Dixon, M., Smart, W. D., and Grimm, C. M. (2004). Say Cheese! Experiences with a Robot Photographer. *AI Magazine*, 25(3):37.
- [Carlisle and Dozier, 2001] Carlisle, A. and Dozier, G. (2001). An off-the-shelf pso. In *Proceedings of the Workshop on Particle Swarm Optimization*, volume 1, pages 1–6.

- [Christianson et al., 1996] Christianson, D. B., Anderson, S. E., He, L., Salesin, D. H., Weld, D. S., and Cohen, M. F. (1996). Declarative camera control for automatic cinematography. In *Proceedings of the National Conference on Artificial Intelligence*, pages 148–155.
- [Christie and Marchand, 2000] Christie, M. and Marchand, É. (2000). Where are they? where do i place the camera? In *Proceedings of the 5th ERCIM-CompulogNet Workshop on Constraints*.
- [Christie and Normand, 2005] Christie, M. and Normand, J. (2005). A semantic space partitioning approach to virtual camera composition. In *Computer Graphics Forum*, volume 24, pages 247–256. John Wiley & Sons.
- [Christie et al., 2008] Christie, M., Olivier, P., and Normand, J. (2008). Camera control in computer graphics. In *Computer Graphics Forum*, volume 27, pages 2197–2218. Wiley Online Library.
- [Crawford, 2004] Crawford, C. (2004). *Chris Crawford on Interactive Storytelling (New Riders Games)*. New Riders Games.
- [Giors, 2004] Giors, J. (2004). The full spectrum warrior camera system.
- [Gleicher and Witkin, 1992] Gleicher, M. and Witkin, A. (1992). Through-the-Lens Camera Control. In *Proceedings of the 19th annual conference on Computer Graphics and Interactive Techniques*, pages 331–340. ACM.
- [Gooch et al., 2001] Gooch, B., Reinhard, E., Moulding, C., and Shirley, P. (2001). Artistic composition for image creation. In *Eurographics Workshop on Rendering*, pages 83–88.
- [Grill and Scanlon, 1990] Grill, T. and Scanlon, M. (1990). *Photographic Composition*. Amphoto Books.
- [Haigh-Hutchinson, 2009] Haigh-Hutchinson, M. (2009). *Real Time Cameras: A Guide for Game Designers and Developers*. Morgan Kaufmann.
- [Halper et al., 2001] Halper, N., Helbing, R., and Strothotte, T. (2001). A camera engine for computer games: Managing the trade-off between constraint satisfaction and frame coherence. In *Computer Graphics Forum*, volume 20, pages 174–183. Wiley Online Library.
- [Halper and Olivier, 2000] Halper, N. and Olivier, P. (2000). Camplan: A camera planning agent. In *Smart Graphics 2000 AAAI Spring Symposium*, pages 92–100.
- [He et al., 1996] He, L., Cohen, M. F., and Salesin, D. H. (1996). The virtual cinematographer: A paradigm for automatic real-time camera control and directing. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, pages 217–224. ACM.

- [Hu and Eberhart, 2002] Hu, X. and Eberhart, R. (2002). Solving constrained nonlinear optimization problems with particle swarm optimization. In *6th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2002)*, pages 203–206.
- [Hu et al., 2003] Hu, X., Eberhart, R. C., and Shi, Y. (2003). Engineering optimization with particle swarm. In *Swarm Intelligence Symposium, 2003. SIS '03. Proceedings of the 2003 IEEE*, pages 53–57.
- [Hwu, 2011] Hwu, W. W. (2011). *GPU Computing Gems*. Morgan Kaufmann.
- [Kennedy and Eberhart, 1995] Kennedy, J. and Eberhart, R. C. (1995). Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 1942–1948.
- [Kirk and Hwu, 2010] Kirk, D. B. and Hwu, W. W. (2010). Programming massively parallel processors: A hands-on approach.
- [Lino et al., 2010] Lino, C., Christie, M., Lamarche, F., Schofield, G., and Olivier, P. (2010). A Real-time Cinematography System for Interactive 3D Environments. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA)*.
- [Liu et al., 2010] Liu, L., Chen, R., Wolf, L., and Cohen-Or, D. (2010). Optimizing photo composition. *Computer Graphic Forum (Proceedings of Eurographics)*, 29(2):469–478.
- [Lok et al., 2004] Lok, S., Feiner, S., and Ngai, G. (2004). Evaluation of Visual Balance for Automated Layout. In *Proceedings of the 9th international conference on Intelligent user interfaces*, pages 101–108. ACM.
- [Mackinlay et al., 1990] Mackinlay, J. D., Card, S. K., and Robertson, G. G. (1990). Rapid controlled movement through a virtual 3d workspace. In *ACM SIGGRAPH Computer Graphics*, volume 24, pages 171–176. ACM.
- [Olivier et al., 1999] Olivier, P., Halper, N., Pickering, J., and Luna, P. (1999). Visual composition as optimisation. In *AISB Symposium on AI and Creativity in Entertainment and Visual Art*, pages 22–30.
- [Parsopoulos and Vrahatis, 2002] Parsopoulos, K. and Vrahatis, M. (2002). Particle swarm optimization method for constrained optimization problems. *Intelligent Technologies—Theory and Application: New Trends in Intelligent Technologies*, pages 214–220.
- [Pedersen and Chipperfield, 2010] Pedersen, M. E. H. and Chipperfield, A. J. (2010). Simplifying particle swarm optimization. *Applied Soft Computing*, 10(2):618–628.
- [Pharr and Fernando, 2005] Pharr, M. and Fernando, R. (2005). Gpu gems 2: Programming techniques for high-performance graphics and general-purpose computation.

- [Phillips et al., 1992] Phillips, C. B., Badler, N. I., and Granieri, J. (1992). Automatic viewing control for 3d direct manipulation. In *Proceedings of the 1992 symposium on interactive 3D graphics*, pages 71–74. ACM.
- [Pulido and Coello, 2004] Pulido, G. and Coello, C. (2004). A Constraint-Handling Mechanism for Particle Swarm Optimization. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 2, pages 1396–1403. IEEE.
- [Ranon et al., 2010] Ranon, R., Christie, M., and Urli, T. (2010). Accurately measuring the satisfaction of visual properties in virtual camera control. In *Smart Graphics*, pages 91–102. Springer.
- [Riguer et al., 2003] Riguer, G., Tatarchuk, N., and Isidoro, J. (2003). Real-Time Depth of Field Simulation.
- [Russell and Norvig, 2003] Russell, S. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Prentice Hall, second edition.
- [Ward, 2003] Ward, P. (2003). *Picture Composition for Film and Television*. Focal Press.
- [Ware and Osborne, 1990] Ware, C. and Osborne, S. (1990). Exploration and virtual camera control in virtual three dimensional environments. In *ACM SIGGRAPH Computer Graphics*, volume 24, pages 175–183. ACM.
- [Weisstein, 2010] Weisstein, E. W. (2010). Least Squares Fitting–Perpendicular Offsets. <http://mathworld.wolfram.com/LeastSquaresFittingPerpendicularOffsets.html>.