UNIVERSITY OF CALIFORNIA

Los Angeles

Towards Intelligent Computational Tools for Virtual Cinematography

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Alan Ulfers Litteneker

2022

ABSTRACT OF THE DISSERTATION

Towards Intelligent Computational Tools for Virtual Cinematography

by

Alan Ulfers Litteneker
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2022
Professor Demetri Terzopoulos, Chair

Virtual cinematography is a fundamental problem spanning a wide range of computer graphics applications. Software for animation, videogames, scientific visualization, and other applications frequently require computational tools capable of automatically determining how to control a camera to capture an image with desired properties. In this thesis, we propose an expressive, controllable, and efficient methodology to virtual cinematography automation that is compatible with both online and offline applications.

By identifying the minima of an unconstrained, continuous objective function matching some desired compositional behaviors, such as those common in live action photography or cinematography, a suitable camera pose or path can be automatically determined using standard search algorithms. With several constraints on function form, multiple objective functions can be combined into a single optimizable function in several ways, which can be further extended to model the smoothness of the discovered camera path with a deformable spline based on an active contour model.

These abstract mathematical techniques are supported by a novel domain-specific programming language, complete with a suite of program analysis and transformation tools capable of automatic differentiation, value range analysis, and program optimization for programs representing run-time specified objective functions, all of which is modularly integrated with Unreal Engine 4 for rendering and user input.

To make this system usable by mathematical non-experts, we explore two approaches.

First, we provide a library of predefined objective functions corresponding to standard photographic and cinematographic compositional rules, complete with recipes for how to combine them to achieve common compositions. Second, we use NLP-derived machine learning techniques on a novel dataset containing annotations on $\sim$ 1M frames from 60 feature films, to attempt to automatically learn objective functions corresponding to real-world compositions.

Finally, these virtual cinematographic techniques are shown to be capable of computing camera paths in either live or scripted scenes with practicable computational costs.

The dissertation of Alan Ulfers Litteneker is approved.

Song-Chun Zhu

Jens Palsberg

Joseph M. Teran

Demetri Terzopoulos, Committee Chair

University of California, Los Angeles

2022

For the people I love,

both near and far...

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This work would not have been possible without the generous support of many people.

Firstly, I cannot express in strong enough terms my profound thanks to my academic adviser, Professor Demetri Terzopoulos. He has spent countless hours over the last 8 years discussing research directions, editing papers, critiquing presentations, instructing, encouraging, and debating with me, without which I would not have even entered the PhD program, let alone completed this thesis. I feel sincerely lucky to have had the opportunity to work under his honest and compassionate tutelage, and will always be grateful for his sustained support.

Secondly, I would like to offer sincere thanks to my doctoral committee: Professor Jens Palsberg, Professor Joseph Teran, and Professor Song-Chun Zhu. Their lively and insightful comments not only helped to strengthen the focus and contents of my thesis, but also steered my research in important directions I never would have considered on my own.

Additionally, I would be remiss if I did not acknowledge my gratitude to the virtual cinematography research community, upon whose excellent work this text builds. While I have been interested in cinematography since adolescence, I was first exposed to virtual cinematography as a serious field of research through the publications of Professor Marc Christie, who has generously shared his time with me at conferences and workshops over the years. As evidenced by the references section, this work also draws considerable inspiration from that of Dr. Christophe Lino, Dr. Quentin Galvane, and Professor Patrick Olivier, among many others. I doubt I would have even embarked upon this research, let alone completed this text, were it not for their estimable contributions to the field.

While working on my PhD, I have been fortunate to collaborate on a number of intriguing research projects with brilliant people, including Professor Tomer Weiss, Dr. Masaki Nakada, Dr. Noah Duncan, Dr. Tao Zhou, Professor Chenfanfu Jiang, and Professor Lap-Fai Yu. These collaborations have exposed me to a variety of fascinating research topics, providing windows into the many wonders of the wider world of computer science research, and preventing me from falling too deep into a pit of monomaniacal myopia around virtual cinematography. I am not only grateful to have had the opportunity to work with such brilliant and dedicated researchers, but also proud of the work we did together.

I have had the pleasure of being surrounded by a wonderful group of labmates during my time at UCLA, including Dr. Ali Hatamizadeh, Dr. Abdullah-Al-Zubaer Imran, and Dr. Garett Ridge, although I have had sadly few opportunities to spend time in the lab since the Covid-19 prompted remote work policies. Before moving to the UCLA Computer Science Department's new building, our lab shared an office with Professor Teran's research group, including Dr. Ted Gast, Dr. Andre Pradhana, and Dr. Qi Guo. The animated discussions and enthusiastic camaraderie of these delightful colleagues always made spending time in the lab a warm and enjoyable experience, for which I am grateful.

One of the most unexpectedly rewarding elements of my time as a graduate student has been in teaching, for which I must thank my supervising instructors, Professor Miryung Kim, Professor Todd Millstein, Dr. Alex Warth, Dr. Alan Kay, and Professor Paul Eggert, as well as the many students that passed through my discussion sections. Even more unexpectedly, I had the distinct pleasure of serving as the Teaching Assistant Consultant (TAC), or Head TA, of the Computer Science Department, helping to train more than 250 graduate students to become teaching assistants over three years. My most heartfelt thanks go to Professor Rich Korf not only for inviting me into the role of TAC, but also for teaching me more about the fundamentals of teaching than I believed possible. I am also grateful to the staff of the UCLA Center for the Advancement of Teaching for their support, as well as to the many graduate students who Professor Korf and I were responsible for training, from whom I learned at least as much as I taught.

I am also grateful to the dedicated staff of the computer science department graduate student affairs office, including Joseph Brown and Steve Arbuckle, without whose patient assistance I would not have been able to navigate to the completion of this degree.

I would also like to thank my colleagues and mentors from before graduate school. During my undergraduate education at Chapman University, I had the privilege of advice and instruction from Professor Erik Linstead, Professor Michael Fahy, and Professor Jurg Walther, among others, without which I almost certainly would not have made it into graduate school. Thanks also to Rob Messer, Damon Tkoch, and the rest of the team at Intellisurvey for inviting me to work on such interesting software development problems with such skilled colleagues, as well as for their flexibility and understanding when I unexpectedly informed them I was considering returning to school. Additionally, I will always be grateful to the many brilliant and dedicated crew members with whom I have worked on film shoots over the years, from whom I learned as much about practical cinematography as any classroom or book could teach. A special thanks also goes to Derek DeRoche, whose infectious enthusiasm for movies and storytelling first spurred me into filmmaking 20 years ago.

Last, but far from least, I am immeasurably, inexpressibly grateful to my family: to my parents, Randall and Rebecca Litteneker, for raising me with unconditional love and tireless encouragement of my education, as well as to my beloved Heather for her enduring love and patience through the unexpectedly long process of finishing this degree. None of this would have been possible without their love and support, for which my gratitude can only be surpassed by my love.

VITA

2018–2021	Teaching Assistant Consultant (Head TA) University of California, Los Angeles Computer Science Department Los Angeles, CA.
2016	M.S. in Computer Science University of California, Los Angeles Los Angeles, CA.
2016–2017	Graduate Student Researcher University of California, Los Angeles Computer Science Department Los Angeles, CA.
2015–2018	Teaching Assistant University of California, Los Angeles Computer Science Department Los Angeles, CA.
2013–2019	Software Developer Intellisurvey, Inc. Ladera Ranch, CA.
2013	Summer Instructor Idyllwild Arts Academy Idyllwild, CA.
2013	B.S. in Computer Science Chapman University Orange, CA.
2013	B.F.A. in Film Production Emphasis in Cinematography Chapman University Orange, CA.
2010–2013	Cinematographer Freelance Los Angeles, CA.

PUBLICATIONS

“Fast and scalable position-based layout synthesis”, T. Weiss, A. Litteneker, N. Duncan, M. Nakada, C. Jiang, L.-F. Yu, D. Terzopoulos, *IEEE Transactions on Visualization and Computer Graphics*, **25**(12), December 2019, 3231–3243. (Date of publication: August 21, 2018.)

“Position-based real-time simulation of large crowds,” T. Weiss, A. Litteneker, C. Jiang, D. Terzopoulos, *Computers & Graphics*, **78**, February 2019, 12–22. Special Section on “Motion in Games 2017.” (Date of publication: October 31, 2018.)

“Virtual cinematography using optimization and temporal smoothing,” A. Litteneker, C. Jiang, *Proc. Tenth ACM SIGGRAPH International Conference on Motion in Games (MIG 2017)*, Barcelona, Spain, November 2017, 10:1–8.

“Position-based multi-agent dynamics for real-time crowd simulation,” T. Weiss, A. Litteneker, C. Jiang, D. Terzopoulos, *Proc. Tenth ACM SIGGRAPH International Conference on Motion in Games (MIG 2017)*, Barcelona, Spain, November 2017, 10:1–8.
MIG 2017 Best Paper Award

“Position-based multi-agent dynamics for real-time crowd simulation,” (poster paper) T. Weiss, A. Litteneker, C. Jiang, D. Terzopoulos, *Proc. ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA’17)*, Los Angeles, CA, July 2017, 27:1–2.

“Fast and scalable position-based layout synthesis”, T. Weiss, A. Litteneker, N. Duncan, M. Nakada, C. Jiang, L.-F. Yu, D. Terzopoulos, *arXiv:1809.10526*, September 2018, 1–13.

“Position-based multi-agent dynamics for real-time crowd simulation,” T. Weiss, A. Litteneker, C. Jiang, D. Terzopoulos, *arXiv:1802.02673*, February, 2018, 1–9.

CHAPTER 1

Introduction

The modern world is increasingly saturated with visual media. Feature length motion pictures, broadcast and internet streaming television, photographs and videos on social media, videogames, and more are enjoyed daily by audiences of various ages around the world. The production of much of this media is an expensive and labor-intensive process. The making of a Hollywood feature film or AAA[1] videogame requires the services of hundreds of artists and technicians over months if not years, at a cost of tens of millions of dollars. Writing plots, designing sets, lighting, editing, acting, and more, all involve significant human effort for even the smallest of projects.

Of particular interest is the task of camera control. The position, orientation, field of view, depth of field, and miscellaneous other settings of a camera affect what elements in a scene are visible and how a viewer will perceive them in the final image. In choosing these settings, artists, designers, and technicians must balance a variety of logistical, aesthetic, and communicative considerations. A change in camera settings can move a movie audience to laughter or tears, decide the effectiveness of a marketing campaign, or determine the fun a player may have playing a videogame.

Often, camera control takes many of its cues from compositional rules common in photography and painting. The "rule of thirds", for example, is held so commonly that many smartphone camera applications provide on-screen guides outlining its use by default. However, many other compositional rules are far more subjective, with practitioners frequently disagreeing on the aesthetic and semantic value of various properties. Would

[1]The classification "AAA," usually pronounced "Triple-A," is somewhat informally used in the videogame industry to refer to high-budget games released by major publishers, analogous to "blockbuster" in the film industry.

an actor look more aesthetically pleasing from one side than another? How big should an object appear in the composition? Should a particular object be visible within an image or absent from it?

Motion picture *cinematography* introduces an entirely new level of complexity to this problem. A viewer's perception of the cinematographer's work is affected not only by the choice of camera settings at any single moment, but also by the relationship between choices made at nearby moments of time. Not only is the desirability of many compositional and aesthetic properties as subjective in cinematography as in photography or painting, but the size of the space of possible camera settings is exponentially larger. As a result, motion picture cinematography is a difficult and specialized task, often requiring extensive manual labor from highly skilled human artists and technicians for each and every shot.

Worse still, videogame players and eSports spectators increasingly expect spectacular visuals, including coherent and evocative camera placement. Unlike traditional motion pictures, there is rarely if ever an opportunity for a human cinematographer to react to these interactive, real time graphics environments. Instead, video game designers and programmers are frequently forced to choose efficiently achievable visual clarity over aesthetic beauty or complex composition in their camera control.

Given all of this, it seems valuable to ask the following question: Is it possible to construct a software system capable of automatically finding a pose, path, or sequence of poses/paths of camera settings that will produce images with desired cinematographic properties? Put another way, can we build an automatic virtual cinematographer?

1.1 Application Domains

In the physical world, camera users are a diverse group, from veteran professionals operating bulky cameras on film sets, to videographers broadcasting footage of sporting events, and school-children capturing their daily activities on camera-phones. An ideal virtual cinematography system would provide efficient camera automation for an equally diverse group of users in virtual spaces, including animators producing emotionally evocative films, game designers specifying engaging camera views for players, scientists visualizing abstruse patterns in 3D data, and enthusiastic laypeople capturing their favorite gaming moments.

While the tools developed in this thesis partially overlap with physical camera control, the focus of this work is on cinematography in *virtual* spaces, where the cameras, actors, and environments exist purely within computer simulations. This is partially a simplifying assumption, as many of the complications of robotic motion control, such as sensor noise and motor delay, can be abstracted away by focusing solely on virtual spaces.[2] However, the automation of *virtual cinematography* remains a difficult problem, with a wide variety of direct applications.[3]

One of the most appealing applications of virtual cinematography rests in videogames and other interactive, real time graphics software. Whenever meaningful user input is unpredictable, programmers and designers will invariably be unable to fully anticipate the state of the game at any given moment. As such, if the images shown to a user are to have certain desired properties, some type of automatic system that is *online* with the execution of the main program loop must be utilized, selecting appropriate camera settings for each frame as it is displayed.

Another group of applications follows the production of 3D animation for films and television, such as those pioneered by Pixar Animation Studios. While the technology and vocabulary may differ, computer animated filmmaking confronts many of the same cinematographic questions as its live action counterpart: What settings should be chosen for a camera to achieve a desired image in a particular scene? However, while live action filmmakers must react in real time to variations in the behavior of their actors, computer animators know exactly what actions their digital actors will take with mathematical

[2]See Section 3.2.1 for a discussion of related work in robotic motion control.

[3]Here, the term "virtual cinematography" is used to refer to cinematography; e.g., the control of lights and cameras, in a virtual space. The term is also used in some computer graphics literature to refer to real time (or near real time) performance capture, animation, and compositing for motion picture production.

precision. As a result, much of computer animated filmmaking treats camera control as an *offline* operation, with camera settings chosen long before any frames are displayed to an audience. Moreover, 3D animation artists frequently design their camera paths iteratively, tweaking camera settings numerous times in an effort to achieve the best result.

The requirements and goals of online and offline applications differ in several important ways. Online system audiences generally expect minimal interruption of their playing or viewing experience, and therefore require any camera control system to display an image matching their desired properties in a fraction of a second. As the content in an online system is to be displayed to a user only once, the decisions made by an online system must be unerringly reliable. By contrast, offline systems need only be efficient and reliable enough to provide useful assistance to an artist who may wish to iteratively modify the camera settings numerous times before finalizing their choices.

Another useful distinction among applications concerns *future knowledge*. The asynchronous nature of many offline applications provides full access to information about past, current, and future events of the scene throughout the decision making process. With undelayed online systems, the viewers' requirement that frames be displayed with a minimum of delay forces any online camera controller to decide where to place the camera using only information available in the current and past scene states. While an experienced artist or clever algorithm may be able to estimate high probability future states, unequivocal future knowledge is categorically unobtainable. Conversely, videogame spectators, as well as players watching replays, are frequently willing to experience a delay of a few seconds between their display and the live scene state. In these scenarios, an online virtual cinematographer can look ahead to future states within the delay interval, utilizing this limited future knowledge as part of its decision making process.

These distinctions between online and offline applications are sometimes unclear. For example, videogame designers must account for a wide array of possible player actions when engineering interactive camera controllers, with complex reasoning about what the player might expect to be shown in different situations. However, after the game is released, the player should feel as though they are in full control of the events and camera views on screen, hopefully incognizant of the designers' heavily engineered automation running behind the scenes. In applications such as these, the final camera behavior results from a combination of both the camera designers' offline decisions and the online interaction of the player/end-user. Depending on implementation, such a camera controller might represent a hybrid between offline and online models.

An important consideration of all these applications stems from the variability in users. An animator seeking camera automation to speed up their work and a scientist in need of help designing aesthetic data visualizations not only have different objectives, but each user type also brings differing skills to the task. An animator or artist might have a unique sense of aesthetics and experience with other camera control systems, but be lacking in code-writing or mathematical analysis skills. Conversely, a software engineer might be an experienced coder, but might have limited abilities in mathematical analysis. In order to be able to service the needs of such a diverse user base, a general purpose virtual cinematography system must carefully balance what users expect with its expectations of the users. For example, a system that expects users to be proficient mathematicians or software engineers would likely exclude much use by artists. Likewise, a system that assumes users all have the same sense of aesthetics might service the needs of novices, but fail to accommodate the individual needs of advanced users.

However, the considered applications of virtual cinematography are not boundless. For the purposes of this thesis, virtual cinematography is primarily concerned with systems capable of controlling cameras in smooth, continuous motions during uninterrupted intervals of time. Beyond the boundaries of this definition lies the task of computational editing, which attempts to automate the decision making of when and how to switch between multiple camera views available during a given scene. Computational editing is an expansive and challenging problem, with a myriad of prospective approaches as diverse and complex as those of virtual cinematography itself. To limit the scope of this work to a manageable boundary, this thesis will remain largely within the boundaries of continuous virtual cinematography, only briefly touching on a few considerations of editing such as the 180° rule.

1.2 System Evaluation Criteria

While the appeal of cinematographic automation is relatively intuitive, specific goals and criteria for the evaluation of prospective systems are not so obvious.

Simple automation may accommodate some simple types of desired camera behavior, but the full range of camera behaviors users may request is enormously diverse. Producing even the simplest of cinematic compositions relies on a host of considerations, including visibility, depth of field, angles between objects and the camera, image position, and more. Worse still, the desirability of various cinematographic considerations is profoundly subjective, as evidenced by the frequency of public disagreements between filmmakers and critics of their work. As such, any singular system intended to serve a diverse group of users must allow for a wide range of *expressivity* in desired camera characteristics.

Orthogonally, any automation system must be relatively *efficient* in order for a user to consider it practically useful. Many online application domains, such as in videogames, may require the camera control system to make its decisions within a fixed window of time, such as in the fraction of a second before the next frame is to be displayed to the user. Other domains, including many offline applications, impose a different sort of time constraint: the system should provide a result for the user at a rate not significantly slower than the user could accomplish unassisted, as any system that takes, for example, an hour to do what a user could alone perform in a minute will not attract frequent usage.

Related to both expressivity and efficiency is *controllability*: A user must be able to quickly and easily control the system such that it will produce desired camera behaviors. Clearly, a user will not be able to satisfactorily use a system the desired camera behaviors are outside the system's available domain of expressivity. Likewise, as the control manipulation required to achieve the user's desired camera behavior becomes more difficult and complex, the amount of user time required will almost certainly increase.

Finding objective means of measuring each of these properties is difficult. A single system might be computationally efficient for one set of desirable camera behaviors and inefficient for another. While one system may be able to support a wider range of expressivity than another, the difference between their supported ranges of expressivity may not contain anything desirable to frequent users. Worst of all, individual users of varying backgrounds may disagree wildly over the controllability of a particular system, and may change their opinions as their level of familiarity with the system changes.

1.3 Approach Overview

Together with the various intended application domains and evaluation criteria, the central question of this thesis might be expressed as: How can a software system be engineered to provide expressive, controllable, and efficient virtual cinematography automation for online and offline applications, with and without future knowledge?

As discussed at length in Chapter 3, there have been a widely varying set of approaches taken to this problem in previous research.

The approach taken here is to formulate the problem of virtual cinematography as continuous optimization, with the user inputting their desired cinematographic properties by specifying a corresponding objective function. Computed approximations of the local optima of this optimization problem correspond to a camera pose or path matching the user's desired properties. Additionally, by carefully balancing the optimization strategies used against the desired precision of the computation, the run time performance of such an optimization system can be tuned to support different levels of efficiency for a variety of application types. Temporal smoothness of the resulting camera paths can then be ensured by regularizing the objective function with a smoothing functional derived from active contour models.

By placing minimal restrictions on the optimization problems allowed as input,[4] this formulation is capable of a high degree of expressivity. However, one important restriction is that objective functions are *vector-based*, meaning that their view of the virtual world

[4] Aside from some technical restrictions regarding the form of the expression discussed in Chapters 4 and 5.

is based upon the position and direction of significant points in the scene. This contrasts with *pixel-based* formulations, which consider the rasterized form of the final image as the basis for the computation.[5]

However, structuring virtual cinematography as an optimization problem implicitly creates a controllability problem: Expressing a desired camera behavior as an objective function requires the user to have mathematical modeling skills, which is an unreasonable expectation to place on many prospective users. This work explores two approaches to mitigating this problem.

1. A library of objective functions corresponding to aesthetic considerations common in traditional cinematography is provided. By using additional predefined mathematical tools to combine objective functions relevant to their desired compositional properties, users with limited technical proficiency can specify complex desired cinematographic behaviors.

2. Alternatively, several machine learning based interfaces, built using data collected from theatrically released feature films, provide mappings from intuitive inputs (such as user chosen frames matching desired compositions) to objective functions compatible with the optimization based virtual cinematography system. While these data-driven interfaces may be more intuitively controllable by novice users, ensuring that this added controllability does not sacrifice system expressivity has proven challenging.

1.4 Thesis Structure

Following this introduction of the virtual cinematography problem and its application domains, the remainder of this document is organized as follows:

Chapter 2 provides a short introduction to cinematography and standard filmmaking techniques, intended to be a brief primer for those lacking a background in filmmaking,

[5] A further discussion of these different approaches appears in Chapter 3.

such as typical computer scientists.

Chapter 3 discusses related work in virtual cinematography and other similar fields, overviewing the varying control styles and algorithmic strategies that have previously been used.

Chapter 4 introduces a mathematical framework for formulating and solving cinematography as unconstrained optimization problems.

Chapter 5 describes the systems engineering underlying the computational tools we implement to support the mathematical framework described in Chapter 4. This takes the form of a small programming language with a variety of program analysis and transformation tools.

Chapter 6 describes a set of methodologies and procedures used to automatically collect egocentric human subject cinematographic data from existing feature films, as well as some preliminary characteristics of data collected.

Chapter 7 presents a library of predefined human authored objective functions, inspired by traditional cinematography practices, as well as the data collected in Chapter 6.

Chapter 8 describes experiments in using machine learning to automatically learn cinematographic objective functions from the data collected in Chapter 6.

Chapter 9 describes a variety of experiments in employing the developed tools for different scenes, objective types, and applications.

Finally, Chapter 10 summarizes the key contributions of the thesis and discusses promising topics for future work.

CHAPTER 2

Cinematography for Beginners

Cinematography is commonly defined as the art and craft of creating motion picture imagery. Etymologically, the term is usually credited as coming from ancient Greek roots meaning "to write with motion" (Brown, 2013, pg. *xiv*). While specific occupational and logistical terminology used can vary, the individual responsible for the cinematography of a film is often referred to as the *cinematographer*.[6] In addition to coordinating with the director and other key creatives, the cinematographer supervises a team of artists and technicians who manage the cameras and lighting, either physical or virtual, to capture or create the images that will appear on-screen.

Cinematography is a complicated and challenging art form, requiring a fine balance between a myriad of aesthetic, communicative, and practical considerations. However, before engaging in a deeper examination of some of these cinematographic considerations, it is worthwhile to begin with a brief tour of some of the basics of motion picture production.

[6]For live action filmmaking in the U.S., the cinematographer is commonly called the "director of photography." In the U.K. and parts of Europe, the responsibilities of live action cinematography were historically split between a "director of photography," who was in charge of the camera, and a "director of lighting," although this separation between roles has been gradually disappearing. Animation studios have historically tasked multiple team members with contributing different specialties, including lighting, lens selection, and color processing, to the cinematography of each shot. The last 25 years have seen some animation studios begin crediting a single head cinematographer or director of photography for each feature film, and in 2014 the American Society of Cinematographers admitted its first animation-specializing cinematographer, Pixar's longtime lighting supervisor Sharon Calahan.

2.1 What is a Camera?

Given that the focus of this thesis concerns controlling cameras, it seems pertinent to briefly address a simple question: What is a camera? Simply stated, a camera is any device capable of capturing images of an environment by sensing light. In physical environments, this light sensing is accomplished by measuring the photons that strike a thin sheet of light-sensitive chemicals or digital chips.[7] In virtual environments, the transport of light through the environment is simulated mathematically with measurements of the simulated light passing through the virtual camera sensor to form the image. In either case, a camera can be thought of as a tool for capturing an image of what a viewer would see if they were standing in the camera's place.

With scarce exception, camera sensors and the images they capture share similar rectangular dimensions. The relationship between these dimensions is usually expressed as a ratio of the width to the height, referred to as the aspect ratio. Digital images captured by sensors are almost always raster images, made up of a rectangular matrix of small square pixels of color/brightness data. The size of that matrix of pixels is commonly referred to as the resolution of the image, and many cameras are capable of capturing images with millions of pixels.

Like any other object, a camera has an orientation in 3D space. This rotation is usually described by cinematographers in three components, with the left-right rotation of the camera referred to as "pan," up-down rotation as "tilt" or "pitch," and rotation around the forward axis as "roll." While these names are slightly different, this pan-tilt-roll rotation description is fundamentally equivalent to the Tait-Bryan angles (e.g., yaw, pitch, roll) common in nautical and aeronautical engineering.

Also obviously, a camera has a position in 3D space. Cinematographers sometimes categorize changes to this position based on their directionality relative to the current

[7]One might present a technically correct argument that light, or electromagnetic radiation, is really a wave within a scalar field governed by quantum mechanics. However, representing light as photons not only accurately models the overwhelming majority of light behavior with which cameras interact, but is also much easier to compute than quantum waves.

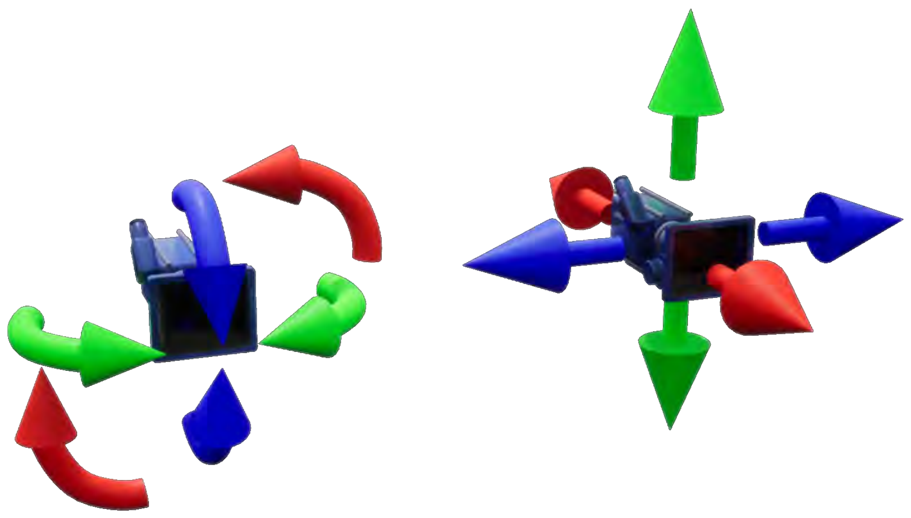


Figure 2.1: Illustration of camera motion directions, with orientation on the left and position on the right: (left) pan movement directions are represented by the green arrows, tilt movement directions by the blue arrows, and roll movement directions by the red arrows; (right) truck movement directions are represented by the blue arrows, dolly movement directions by the red arrows, and crane movement directions by the green arrows.

orientation of the camera, with movements that are left-right referred to as “truck” movements, forward-back as “dolly” movements, and up-down referred to as “crane,” “jib,” “boom,” or “pedestal” movements. The computer graphics community usually instead refers to the “right” or “left” direction, the “up” direction, and the “forward” or “look” direction of a camera.

In order for the light bouncing around the environment to form a coherent image on the sensor, a lens system is placed between the scene and the sensor. Different camera lenses can work in many different ways, from a simple pinhole to multi-element compound optics, and are worthy of considerably deeper discussion than appropriate in this text. In simple terms, most camera lenses can be understood as modified pinholes. Suppose there is a rectangular cuboid box made of an opaque material with an infinitely small opening, an ideal pinhole, on one side. As light moves in straight lines absent extreme conditions,[8]

[8]Like strong electromagnetic fields or a significant relativistic curvature of space-time, neither of which are usually desirable on a movie set.

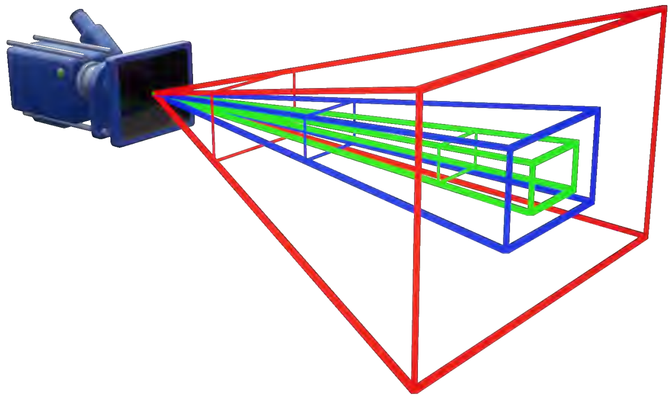


Figure 2.2: The green, blue, and red lines in the lower image visualize the region of the scene visible to the camera with increasing FOV's and decreasing focus distances.

all light that passes through the opening and hits a particular point on the opposite side of the box must come from the same direction in the scene, resulting in the projection of a clear image of the scene. However, while an infinitely small pinhole will allow no light to pass through, making the aperture larger will cause the projected image to become blurry, as points in the scene will now project to overlapping regions on the image plane. Practical lens systems solve this problem by placing optics (e.g., shaped glass or mirrors) around the aperture that ideally[9] focus light from a point a set distance in front of the camera on to a point on the sensor plane, regardless of aperture size.

This set distance at which points appear in focus is usually referred to as the focus distance, and most lenses allow this focus distance to be adjusted. Objects not at this set focus distance will appear blurry, with objects further from this set distance appearing blurrier. The range of distances at which objects appear acceptably sharp is commonly referred to as the depth of field of the lens. Depth of field is usually mathematically computed using a metric called circle of confusion, which models the diameter of the circle in the image to which a point in the scene would be projected.[10]

[9]In practice, imperfections in the lens as well as electromagnetic interactions with the environment enforce a nontrivial limit, called the diffraction limit, on the achievable sharpness at a specific distance.

[10]For an ideal symmetric lens, the circle of confusion $c = \frac{Af|S_2 - S_1|}{S_2(S_1 - f)}$ for an object at distance S_2, where S_1 is the focus distance, A is the aperture diameter, and f is the focal length of the lens. While this works perfectly for virtual lenses, most physical lenses are not symmetric and therefore require more

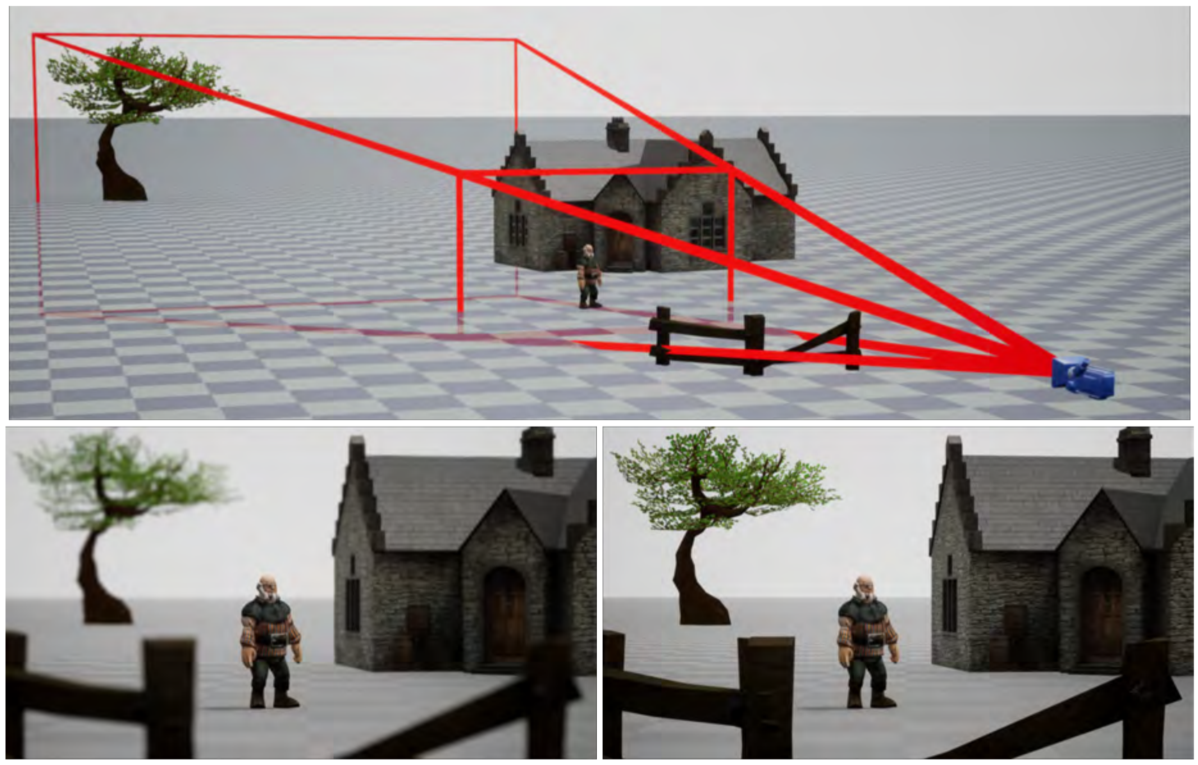


Figure 2.3: Illustration of a rectilinear perspective camera view of a scene. The general layout of the scene is shown in the upper image, where the camera frustum is visualized by the red lines. Two different frames from this same camera view are shown below, with the left image demonstrating shallow depth of field (e.g., large aperture), and the right image showing a deep depth of field (e.g., small aperture).

Like pinholes, most modern camera lenses are rectilinear, projecting straight features in the environment as straight lines in the image. With a rectangular sensor, rectilinear lenses allow for a frustum shaped region of 3D space to be observable in the 2D image. When measured angularly relative to the apex, the size of this visible region is referred to as field of view, commonly abbreviated to FOV, and can be measured either horizontally, vertically, or diagonally relative to the camera's orientation, with the aspect ratio allowing conversion between any of these measurements. In optical terms, this FOV is controlled[11] by a combination of the focal length of the lens system and the size of the sensor.[12] A

complicated mathematics to model accurately.

[11]FOV$= 2 \arctan \frac{d}{2f}$, where d is the length of the diagonal of the sensor rectangle, and f is the focal length of the lens.

[12]In physical cameras, the focal length of the lens is more commonly used. FOV is more common in virtual cameras, especially when no sensor size is explicitly specified, although many feature rich software

change to this field of view is commonly referred to as “zoom.”

While most real and virtual cameras can carry a variety of other settings that affect the final image,[13] a camera can be considered to be fully defined for the purposes of this text by the properties already specified, which can be summarized as follows:

- position in 3 dimensions,
- orientation in 3 dimensions,
- focal length (or FOV),
- focus distance,
- aperture size,
- and aspect ratio.

2.2 Modern Filmmaking Processes

More than a century of motion picture production has not only resulted in more complex and diverse film content, but also increasingly complicated and varied filmmaking processes. However, any motion picture production can generally be divided into three sequential phases: pre-production, production, and post-production.

The **production** phase is the primary phase in which motion picture imagery is captured or created, for example when there are cameras pointed at performers in a live action production or animators giving their characters the illusion of life. Given the large numbers of people and volumes of equipment commonly required, time in this production phase is often extortionately expensive and highly restricted.

As a result, a huge amount of effort is spent preparing before production of a scene begins, a period referred to as **pre-production**. Often, pre-production for a film runs

packages support both.

[13]Including sensitivity, resolution, frame rate, exposure time, brightness, contrast, color temperature, and more.

longer than production itself, with every opportunity to save time or money in production carefully weighed against their contributions to the communicative intent of the project as a whole.

Even after pre-production and production have been completed, there is usually significant work left to be done, whether it is extensive editing, music scoring, and visual effects or merely tearing down sets and delivering digital files. This final phase is commonly referred to as **post-production**.

While these three phases are relatively ubiquitous, the boundaries between these phases are not always clear. For example, production may begin while many important decisions associated with pre-production, such as who to hire or where to film, are yet to be made. Similarly, the machinery of post-production often begins long before production has finished, with editors and visual effects artists often commencing work on footage as soon as it is available.

However, below the thin veneer of these three common phases lies an enormous diversity in process between productions. While some of this variability in process stems from cultural and institutional differences between groups of filmmakers, the most significant differences stem from the form and format of the intended exhibition of the resulting motion picture. The variations between the production of a feature film for theatrical distribution, an episode of a television series, a live news broadcast, coverage of a sporting event, a documentary, or almost any other type of motion picture can be roughly categorized by three criteria:

(1) whether the events shown on screen are **scripted** or **unscripted**,

(2) whether the images are intended to be displayed **live** or **recorded** for playback to each audience,

(3) and whether the actions portrayed are the result of capturing live **performances**, often called live action, or the result of **incremental** animation.

2.2.1 Scripted Filmmaking

The pre-production of a scripted motion picture, whether a big-budget Hollywood film, a television advertisement, a training video, or a YouTube short film, usually begins with the writing of a script or screenplay containing high level textual descriptions of the events and dialogue to be depicted in the film. This screenplay is the primary source for specifying whether the resulting film is to tell a story, sell a product, raise awareness of an issue, provide critical information in an accessible manner, or whatever the desired communicative intent might be. With rare exceptions, screenplays do not describe any specifics as to how these events should appear on-screen.

Before production can begin, the principal artists and technicians working on a project must decide how to break down the screenplay into visual elements to be displayed on screen. How should the sets, costumes, and props look? Where and when should the actors move in each scene? Should certain colors be used or omitted to evoke certain emotions or themes? What sorts of camera angles and lighting should be used to convey the communicative intent of each scene? Worst of all, how should all of the creative goals of the project be balanced with its budgetary and logistical constraints? This pre-production decision making process is enormously complex and, at the macro level, well beyond the scope of this paper.

However, once enough of these decisions are made, production will begin. For live action productions, once sets are built, equipment is procured,[14] and actors are rehearsed, production consists of putting everyone together in the same space and turning on the cameras. This live action process is inherently performative, with the real-time action and reactions of actors, camera operators, and lighting technicians collaboratively contributing to the resulting motion picture imagery.

The production phase of animation, whether 3d computer generated, 2d hand drawn,

[14]Tangentially, the full purchase price of a new cinema camera package, including with a full set of interchangeable lenses and other support hardware (e.g., batteries, view-screens, recording media), can easily exceed $100,000 in 2022 currency. With incredibly rare exception, productions prefer to rent camera equipment for a few hundred dollars a day rather than buy.

stop-motion, or some hybrid, traditionally takes a different direction. Most animation starts by creating a rough version of each scene in the film, often with a very low framerate, specified at important moments called key frames, and crude drawings/models. This rough version is incrementally improved, its framerate raised[15] and its drawings/models refined, until a desired final quality is reached.

However, some forms of animation can blur the line between iterative and performative practices. For example, motion capture technology can allow character animation to be specified through an actor's performance in real-time, but often requires extensive iterative modification after the fact to help convert a human actor's motions to those befitting the fantastical creature being portrayed.

Moving on to post-production, the goal is to take the footage created in production and combine in a way that matches the communicative intent of the project. In recorded productions, this usually means that recorded footage must be edited and colored, visual effects rendered and composited, music and sound effects chosen and mixed, and more before the motion picture can be shown in theaters, broadcast on television, or streamed on the internet. Additionally, the fact that production is done offline from editing and the other transformations of post-production means that multiple attempts, usually called takes, of each shot can be made in production, with the best bits from each pieced together in editing.

Unsurprisingly, live broadcast productions must achieve all of these same tasks in real time, delicately walking a tightrope between what can be accomplished at a moment's notice and what best matches the creative vision of the filmmakers. Worse still, as the production is broadcast live, there are no opportunities to redo anything in the event of human or technical errors, often heavily incentivizing caution from those uncertain of the limits of their capabilities. However, because the project is scripted, these live editors and technicians can make reasonably informed guesses as to what the performers will do from moment to moment.

[15]This process of creating frames in-between already animated frames is often called in-betweening in hand drawn animation.

Note that, in this taxonomy, unscripted and scripted live broadcast incremental animation is not a practicable form, as the iterative refinement part of the workflow is not readily compatible with live broadcast.

2.2.2 Unscripted Filmmaking

While scripted productions begin with at least a basic understanding of what events should appear on-screen, unscripted projects, such as documentaries, coverage of sporting competitions, or talk shows, aim only to capture or portray events as they unfold, with little to no knowledge of what those events might be. This is not to say that unscripted films cannot begin without a communicative intent: broadcasters of a football match want to show the drama unfolding between players on the field, documentarians may know what answers they are hoping to record before starting an interview, and talk show hosts often have sets of general conversational topics prearranged.

However, the lack of foreknowledge of on-screen events fundamentally limits the pre-production priorities of unscripted filmmakers to merely creating a pool of available equipment and procedures for capturing as much of whatever happens as possible. Instead, much of the creative decision making surrounding how to achieve the communicative intent is shifted to production and post-production, with camera operators reactively capturing whatever imagery best matches the tone of the moment, and editors deciding how to string that imagery together to achieve the communicative intent.

What about motion pictures stemming from videogames, for example broadcasts of esports events? In this categorical breakdown, videogame visuals represent nothing more than unscripted, live broadcast, live performance productions. The complication arises from the fact that the artists and technicians responsible for real-time decision making are algorithmic rather than human. And thus we return to the question raised at the beginning of this work: can humans translate their skill at cinematography, editing, or any other element of production into an algorithm?

2.3 Cinematography in Practice

An element critical across all modern filmmaking processes is cinematography. In scripted and unscripted, live-action and animated motion pictures, the audience sees the drama, dialogue, and data presented on screen quite literally through the lens of the camera. Determining how to move the camera to properly convey the desired communicative intent of each beat of every scene is therefore a vital but challenging task.

For the purposes of this work, we will assume that the key creatives of the project, such as the director and cinematographer, have already decided what types of cinematographic compositions best suit the desired communicative intent for each moment. These individuals have certain images, motions, and compositions already in mind before meeting with their crew.

However, cinematography is generally a collaborative endeavor. Live action cinema cameras are large and unwieldy machines,[16] commonly requiring hands-on management from 2-4 technicians[17] when recording. Cameras in animation, whether physical or virtual, are often preliminarily set up by one artist,[18] before having their settings repeatedly refined by a sequence of subsequent animators and camera artists.[19]

Therefore, an important question is how complex cinematographic decisions are communicated between crew members, so that what is captured by the cameras matches the original cinematographic intent.

[16]For example, the Arri Alexa, a digital cinema camera currently popular among professional filmmakers, together with a compatible lens, battery, and recording media, usually weighs 30-50 pounds (13.6-22.7 kg), and is 2-4 feet (0.6-1.2 m) in length (Goi, 2013). As a fun comparison, that is in roughly the same size and weight category as a fully grown medium-sized dog, such as an Australian Shepherd.

[17]Such a camera team is commonly composed of a camera operator, who customarily controls the orientation of the camera, and one or more camera assistants, normally responsible for the camera focus, aperture, and zoom, among other logistics. Some camera movement apparatuses, such as camera dollies, additionally require another crew member to precisely push the camera (and also sometimes the camera team) around the stage on the apparatus, such as a dolly grip pushing a dolly.

[18]Usually referred to as a layout artist.

[19]In many 3D animation pipelines, camera artists are among the last to touch the camera settings, following a long line of layout artists and character animators.

Frequently, filmmakers communicate their cinematographic intentions in the form of storyboards and shot lists. **Storyboards** are illustrations or diagrams that visually show the compositional arrangement of key actors and props in the scene. While the technology and styles used vary, they can generally be read in a similar manner to a cartoon or graphic novel, with consecutive panels describing either motion within a shot or consecutive shots separated by edits. **Shot lists** provide similar information in the form of sequential textual descriptions, often using abbreviations and other shorthands for efficient communication.

Despite stylistic and terminological variations, methods such as storyboards and shot lists commonly communicate similar cinematographic information.

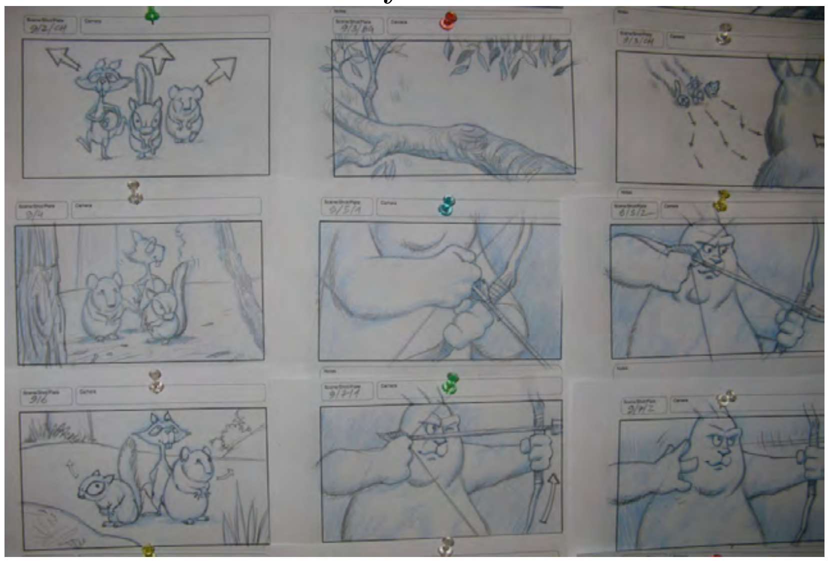
- Perhaps the principal property in cinematography requiring specification is that of **visibility**: what actors and props should be visible within the frame? Just as important as determining what should be visible is determining what should be hidden in the frame. In dialogue scenes, a common question is whether a frame of one actor speaking to another actor should show both actors or only one. A frame of an actor alone is often referred to as *clean*, while a frame showing part of the back of the actor being spoken to is said to be *dirty* (Brown, 2013, p.21).

- Closely related to visibility is **frame position**: roughly where in the frame should each visible object be? Frame position is often expressed in relative directional terms, with objects being described as being described as being left, right, above, or below other visible objects. For many scenes between two people, frame position is closely related to the line of axis, with shots of each actor describing them as being on the left side of the frame looking towards the camera's right, or right looking left. This directionality is often referred to as *frame* or *screen direction* (Brown, 2013, p.81-82).

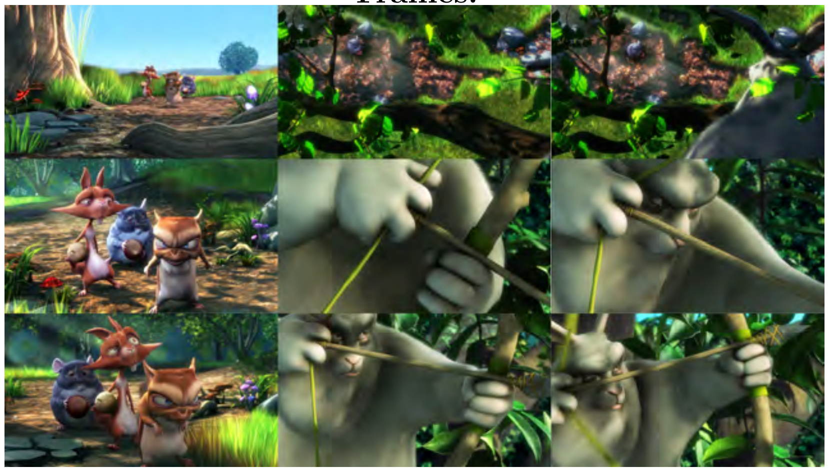
- The size of an actor or prop within the frame is commonly described as **shot size**. The amount of space an object occupies in the frame generally reflects its importance in the scene, a relationship commonly referred to as the Hitchcock Rule (Brown,

Synopsis: A trio of rodents, already established as notably unpleasant bullies, walk into a forest. Buck, a large bunny and the story's protagonist, watches the forest floor below from a perch high in a tree. Upon seeing the rodents pass along a trail below, Buck draws a bow and arrow. First seeming to aim the dangerous weapon at the rodents below, Buck adjusts his aim high into the treetops before loosing the arrow. (After the events shown here, the fired arrow is shown to cut a vine, releasing a swinging log that comically whacks into one of the rodents.)

Storyboards:



Frames:



Shot List:

1. Extreme wide shot (XWS) of the rodents entering the forest by approaching the camera.
2. High angle over-the-shoulder (OTS) of Buck towards an extreme wide shot (XWS) of the rodents walking below, as Buck watches the rodents from high in a tree.
3. Wide shot (WS) from ground level of the rodents walking through the forest, seemingly unaware of the threat above.
4. Profile close up (CU) on Buck as he draws back the bow aiming downwards, seemingly towards the rodents below.
5. Medium shot (MS) on rodents continuing to walk through the forest, still unaware of Buck.
6. Profile CU of Buck as he swings the bow upwards before loosing the arrow.

Figure 2.4: Synopsis, shotlist, storyboards, and corresponding frames from a scene in *Big Buck Bunny* (2008).

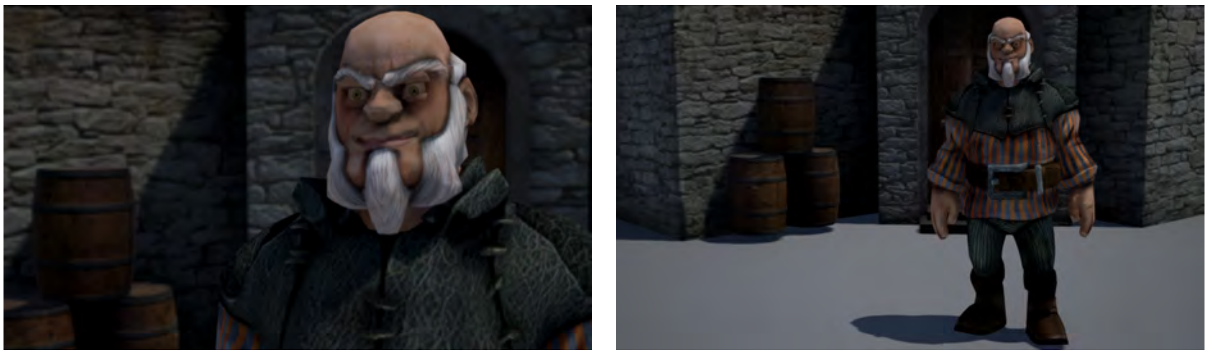


Figure 2.5: Examples of a close-up shot (left) and a wide shot (right).

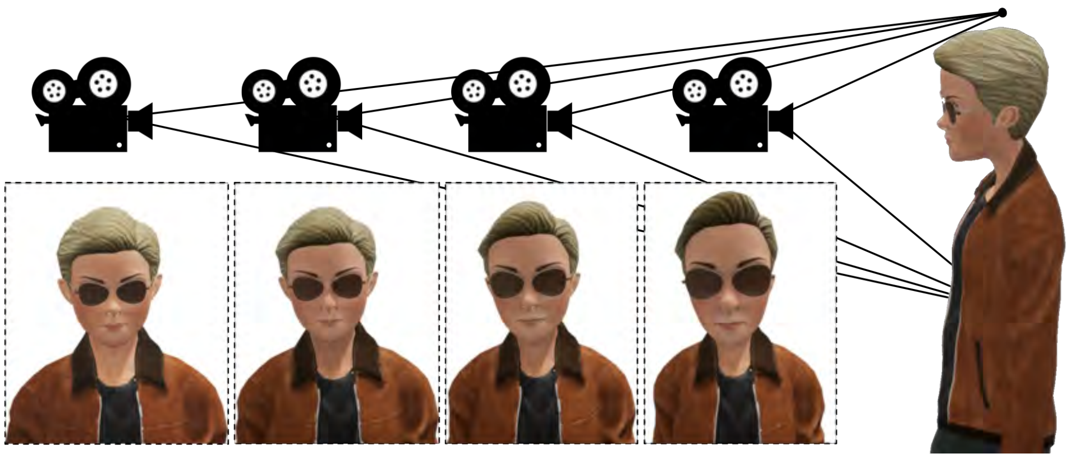


Figure 2.6: Examples of how varying camera distance can modify the appearance of the same shot size.

2013, p.29). For human subjects, this is usually expressed in terms of how much of a particular person is on-screen. For example, a view of a person's entire body, commonly called a wide or long shot (often abbreviated *WS* or *LS*), is frequently used to establish the relative geometry of objects in a scene, while a view of only their head and shoulders, commonly called a close-up shot (abbreviated *CU*), is often used to show dialog (Bowen and Thompson, 2013, p. 8-11). As illustrated in Figure 2.6, an actor can be depicted at the same shot size from a variety of distances by modifying the camera field of view.

- The angle from which an audience is shown a particular object can dramatically change the way that an object or character is viewed. This **relative angle** characteristic can generally be divided into two categories, vertical and profile angles,

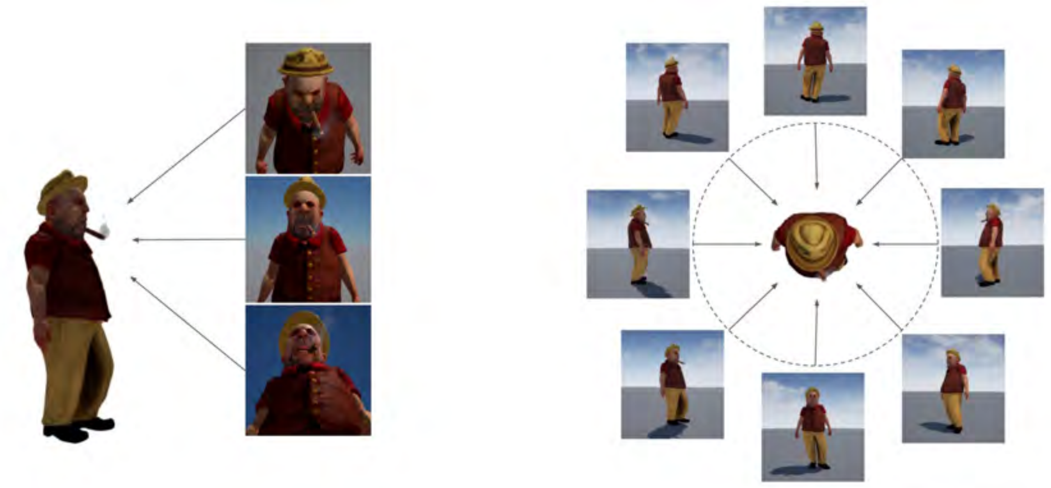


Figure 2.7: Examples of vertical angles (left) and profile angles (right).

examples of which are shown in Figure 2.7.

- Objects, and people especially, tend to appear more powerful, intimidating, or ominous when viewed from below, and conversely more weak, frightened, or small when viewed from above. This relative **vertical angle** is often used to demonstrate the changing relationships between characters in a story, making each relative rise or decline visually clear to the audience (Bowen and Thompson, 2013, p. 34–39).
- The way in which people are perceived is strongly affected by the angle of view relative to the direction in which they are looking, or **profile angle**. In general, characters that are viewed directly from the front are felt to be more connected to the viewer than characters viewed in profile, mimicking the way humans tend to face directly towards things upon which they are focused (Bowen and Thompson, 2013, p. 40–43).

However, storyboards and shot lists are often incomplete, relying upon the knowledge and experience of those executing the shots to achieve comprehensible and aesthetically pleasing results. Unlike the cinematographic properties already specified, these informal rules are commonly held as being applicable for consideration for any frame without explicit specification.

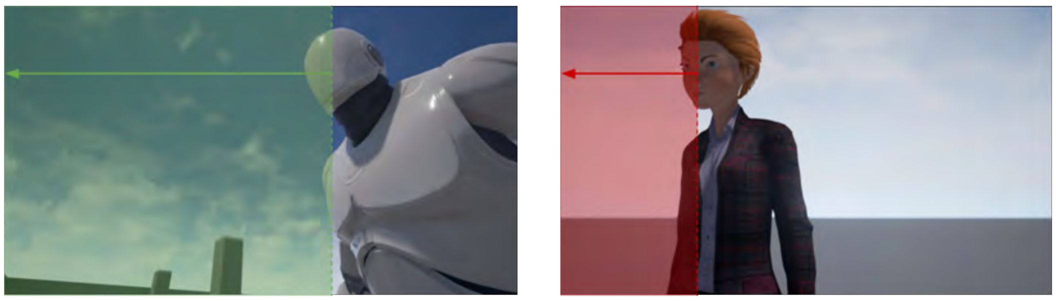


Figure 2.8: Example frames with (left) and without (right) adequate look space.

- For over a hundred years, a standard rule of thumb in photographic and cinematographic composition has been to place important objects near the third lines, both vertical and horizontal, in the frame. The most important objects in the scene are often framed nearest the intersections of these lines (Brown, 2013, p. 51). While this **rule of thirds** is certainly popular, it is far from inviolable. The curious need only leaf through a fashion magazine or randomly pause a feature film to discover compelling images where important figures are not positioned along the third lines of the frame. However, there appears to be little published consensus regarding criteria for exactly when and how to break the rule of thirds.

- A common technique used to balance a shot is to give substantial space between a character and the frame bound in the direction they are looking, a property that has been given numerous names in the cinematographic literature, including "**look space**", "lead space", "nose room", "action room", etc. When this space is not provided, as is shown in Figure 2.8, viewers tend to perceive the character as psychologically on edge or boxed in, a property that can sometimes be utilized to interesting compositional effect (Brown, 2013, p. 52).

- In many circumstances, some parts of an object or actor should be inset some amount from the edge of the frame. This is most commonly evident with human heads, as too much room between the top of the head and the edge of the frame causes the subject to appear small and insignificant, while too little causes the subject to appear restricted and boxed in. This property is commonly referred to

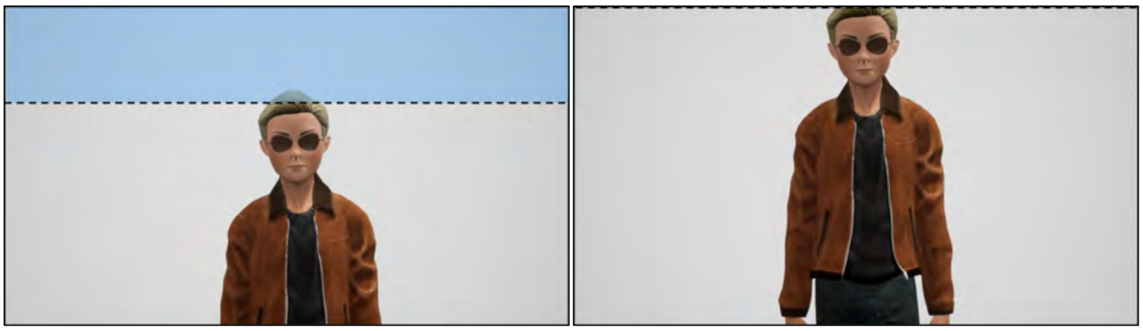


Figure 2.9: On the left is a frame where the character has too much headroom, as highlighted in blue. On the right is a frame where the character has too little headroom, with the edge of the frame cutting through their hair.

as **headroom** or **head room** in traditional cinematography (Brown, 2013, p. 52).

- An obvious consideration worth mentioning is that, with almost no exceptions, cinematographers and audiences alike expect the camera to not collide with any objects in the scene. In live action cinematography, any collision can damage the sensitive electronics and machinery of the expensive camera, as well as produce a jarring jolt to the perceived motion.

Additionally, other properties become relevant when considering sequences of shots separated by edits. While watching a sequence of shots intended to portray spatially and chronologically continuous events, a viewer must be able to quickly determine how characters and props from one shot correspond to those of a later shot. While many of the broader considerations of continuity editing are outside the scope of this work, one is vital.

Arguably the most significant and well studied continuity editing rule is the aforementioned **line of axis**, also called the 180°rule. In any scene between multiple actors or objects of focus, the camera almost always stays on one side of a line[20] drawn connecting the groups, as illustrated in Figure 2.10. As long as the camera does not cross this line, an actor that is to the left of another actor will remain on the left, while an actor on the right will remain on the right. If the camera crosses this line, the relative horizontal

[20]It would be more geometrically accurate to call this 3d boundary a plane than a line, but every source I have read continues to call it a "line," probably for historical consistency.

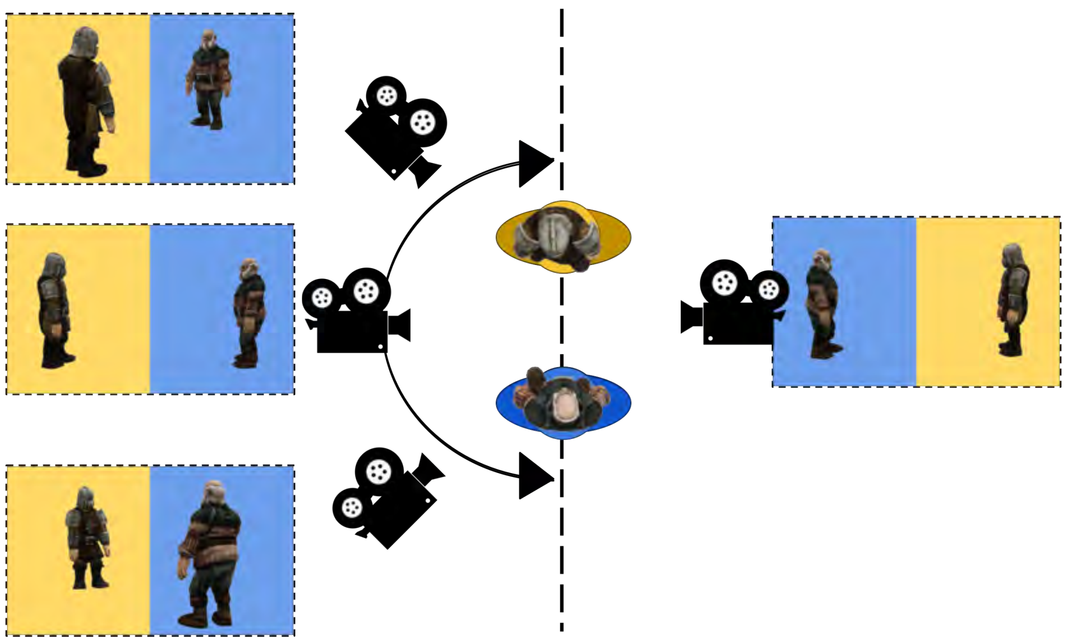


Figure 2.10: A diagram illustrating the effect of the line of axis. The central dotted line that passes through the two characters is the line of axis for this scene. All camera angles on the left side of the line of axis show the characters on the same sides of the screen, with the yellow figure on the left and the blue figure on the right. However, any camera angle on the right side of the line of axis will show the characters reversed, with the yellow figure on the right and the blue figure on the left.

positioning of objects in the frame will suddenly and jarringly invert, with actors often appearing to switch places, something commonly called a jump cut. As a result, a key question is often between whom the line of axis is between, and on which side the camera should stay (Brown, 2013, p.80-85).

One of the features of these properties that I hope is clear is their immense irregularity. Even the most sacrosanct of these rules are commonly held to be eminently violable, with conditions for their allowable violation somewhere between subjective and inexplicable for even the most accomplished of filmmakers. While the semantics of the various properties outlined in the section above are useful shorthands for modern cinematographic expression, cinematographers frequently disagree as to the relative importance and applicability of rules such as these. For a general purpose virtual cinematography system to be useful to even a small group of real users, some mechanism for the expression of individual cinematographic preferences must be supported.

2.4 On Editing and Broader Semantics

It can be tempting to imagine, given the relative simplicity of the cinematographic rules outlined above, that a similar set of rules could be specified for the semantics of sequences of shots. Unfortunately, determining anything objective about the semantics of editing represents an obscenely challenging task. While film-going audiences are often able to agree on the meaning of a sequence of shots, they are rarely able to agree as to how specific components contribute to the meaning of the whole.

A simple example of this can be found in the famous Kuleshov effect. In the 1920s, filmmaker and film theorist Lev Kuleshov produced a set of three experimental short films. Every film began and ended with the same close up shot of the expressionless face of a well known actor of the era, Ivan Mozhukhin,[21] but each film cut to a shot of a different scene in the middle: a girl playing with a stuffed animal, a steaming bowl of soup, or a body in a coffin. When Kuleshov exhibited any one of these films, audiences universally lauded Mozhukhin's performance regardless of which film they had seen, celebrating the subtle joy with which he watched the playing girl, the pensive hunger with which he viewed the soup, and the deep sorrow with which he viewed the coffin. Kuleshov concluded that these profoundly different perceptions of the same expressionless face could only be explained by the viewers naturally perceiving the meaning of each shot from a combination of the shot itself and the shots that had been edited to surround it (Cook, 2004, p.118-120).

Despite considerable efforts from scientists and films scholars over the last century, the source and fundamental nature of these editing effects are remarkably poorly understood. Medical researchers conducting fMRI experiments only succeeded in statistically verifying the existence of the Kuleshov effect in 2006, although the specific neurological phenomena responsible remain largely unknown (Mobbs et al., 2006). A handful of authors, such as

[21]Sometimes spelled Mosjoukine or Mozzhukhin in English language texts. As an ally of several prominent Tsarists, Mozhukhin fled Russia in response to the 1917 revolution, and was living in Paris when Kuleshov's experimental films were being made. Kuleshov supposedly used footage from one of Mozhukhin's earlier films, the prints for which had been abandoned during Mozhukhin's flight to France. Unfortunately, there are no known surviving copies of either Kuleshov's shorts or many of Mozhukhin's films.

Arijon (1991) and Katz (1991), have attempted to pen practical guides on the selection of sequences of shots to reach a specific semantic intent, though a close reading of their works finds little consensus.

What limited agreement exists centers around **continuity** editing, a general strategy of editing shots together so that they coherently appear to refer to continuous regions of time and space. One rule specified in the last subsection, the 180°rule, has its roots in continuity editing. However, rules such as these do not provide an opportunity to communicate any message to the viewer, instead simply guarding against cuts that will strain the perception of spatial continuity.

Beyond the boundary of continuity editing, the semantics of film editing remain enduringly inscrutable. Just as a practiced speaker of English, Mandarin, or Hindi can easily predict the various ways a fragment of speech can be interpreted, skilled filmmakers seem to be able to intuitively identify the meanings an audience is likely to glean from an edit or shot choice, and leverage this skill to select edits that best match their desired communicative intent. What guides this ability, and whether it is something modellable in a computer, is surprisingly difficult to analyze.

Early film scholars, such as Kuleshov, argued that the language of film could be analyzed according to static formalisms, such as montage theory, though they themselves were not particular consistent in following their own advice. Some film scholars, such as Metz (1991), have attempted to linguistically formalize film through semiotics, although the frameworks they propound are remarkably unwieldy for anything but abstract analysis. Some more recent film scholars have theorized that the observed meaning of a film emerges from some combination of the images/sounds exhibited and the psychological state of the audience, which is in turn impacted by the simultaneous presence of the film's content within a multitude of cultural contexts. While facilitating many types of fascinating analysis and argument, this psychoanalytical paradigm provides little additional insight into how filmmakers make decisions.

Regardless of where the truth lies in this wild array of hypotheses, it seems as though the language of film is neither static nor uniform. Feature films and music videos, for example, frequently exploit divergent cinematography and editing styles, with each style undergoing continuously dramatic evolution from year to year. Internet websites and forums constantly feature contradictory opinions about how good the latest film or television episode was, with much of that disagreement stemming from a lack of consensus on exactly what meaning was contained in the work. As a result, attempts at automatic edit comprehension or computational editing[22] are fraught with difficulty, and have therefore been resigned to the dark expanse of regions unexplored in this text.

[22]See Section 3.2.3.

CHAPTER 3

Related Work

Virtual cinematography is certainly not a new problem. From simulated spacecraft to household robots, medical imaging to film production, and videogames to scientific visualization, cameras have been and continue to be an integral part of many virtual and physical systems that require or can benefit from automation.

As befitting a fundamental problem with wide-ranging implications, virtual cinematography has been subject to diverse directions of research under a variety of names. As this work is focused on camera control for cinematographic applications in virtual spaces, some effort has been taken to use the term "virtual cinematography" consistently throughout this document. However, similar problems have been referred to as virtual camera control by Abdullah et al. (2011), virtual camera planning by Pickering (2002), and viewpoint selection by Vázquezz et al. (2001), among a variety of other names. Much of this diversity stems from the differences between the backgrounds of the researchers and their intended applications for developed tools.

While visibility and illumination problems have long held interest in analytical mathematics, the first work to explore camera control explicitly for 3D computer graphics[23] was Blinn (1988), introducing several vector-based algebraic control schemes now ubiquitous in computer graphics, such as the LookAt matrix. Somewhat more recently, two surveys of the field have been published, specifically Christie et al. (2005) and Christie et al. (2008), and much of the terminology and organization of this chapter is built upon their excellent work. Another good overview is Haigh-Hutchinson (2009), an equally comprehensive

[23]Specifically for what Blinn amusingly called his "space movies." At the time, he was employed at the Jet Propulsion Laboratory, where he created many pioneering renderings of spacecraft in flight.

but somewhat more practical overview of interactive camera control for videogames.[24] While these surveys and practical overviews are well researched and thorough works, a considerable volume of research has been published since their publications. Additionally, none of these works provide a meaningful taxonomy that allows for widely differing works to be meaningfully compared, instead enumerating different categories of applications served and approaches taken with limited clear interrelationship.

As already mentioned,[25], there are many different directions of research within just the computer graphics community. Videogames, medical imaging, animation, scientific visualization, and more prescribe distinct and sometimes contradictory demands on the form and formulation of virtual cinematography systems. Some of the differences between these applications were discussed in Chapter 1.1, including online/offline computation, the degree of available future information, and variability of desired camera behaviors.

However, this assortment of approaches can be understood as different answers to a single question: to achieve a desired image at one or more moments of time, how can a computer decide where to place a camera based on the positions and orientations of significant objects in a 3D virtual scene?

This way of posing the problem of virtual cinematography is relatively broad, admitting a wide variety of different types of research under its umbrella of related work. However, it does place three useful constraints on the body of work considered related:

1. the 3D motion (e.g., position in $\mathbb{R}^3$, rotation in $SO(3)$, lens/sensor settings)[26] of a virtual camera is being controlled

[24]This book was primarily written by Mark Haigh-Hutchinson, a veteran video game developer who specialized in camera control systems in the latter part of his career. Sadly, he unexpectedly passed away in 2008, after which his unfinished manuscript was edited into the published book by several of his former colleagues. To my knowledge, Haigh-Hutchinson (2009) is the only comprehensive book on practical camera control systems to be published to date.

[25]Including in sections 1.1 and 1.3, as well as earlier this section.

[26]Note that any subset of these parameters satisfies this definition, as well as any alternative representation convertible to parameters like these, such as spherical positional coordinates.

2. within a wholly virtual (i.e., entirely modeled within a computer simulation) scene,[27]

3. and the computation of its control is vector-based (i.e., uses only the positions and orientations of objects of interest in the scene).[28]

With these more precise criteria for related work specified, how should a reader interested in virtual cinematography go about understanding the multiplicity of related work?

3.1 Taxonomy of Related Work

The body of work directly related to the criteria outlined at the beginning of this chapter is large both in volume and diversity of approaches. As many of these related works introduce novel or conflicting terminology and priorities, understanding these works in a single context can prove challenging.

Fundamentally, a virtual cinematography system is little more than a collection of algorithms capable of automatically computing camera settings corresponding to a user's specified behavior. However, different systems can be capable of widely varying ranges of behaviors for a variety of reasons. Disparate algorithms not only have different capabilities, but different user interfaces can also expose different portions of those capabilities to users, which diverse users may find more or less useful to their specific needs and preferences.

Following this basic conceit, virtual cinematography systems can be analyzed through three criteria.

[27]While several robotic systems are mentioned in the following sections, such as Bonatti et al. (2021); Huang et al. (2019a,b, 2021), these systems first consider cinematography in the virtual world constructed from analyzing their sensor data, then use the virtual cinematography solution to compute what robotic motions to compute. In this sense, the purely virtual portions of these systems still satisfy this requirement.

[28]Several approaches, such Olivier et al. (1999); Halper and Olivier (2000) that require algorithmic analysis of rendered pixels are mentioned here, especially in the context of visibility computation. However, these systems do not consider the material properties of the objects in the scene, instead using the rendering pass as an accelerated form of geometry query functionally equivalent to raycasting. More philosophically, computing with enough points on the surface of some geometry could easily approximate the level of detail of rendering while satisfying this definition, as long as no material properties are considered.

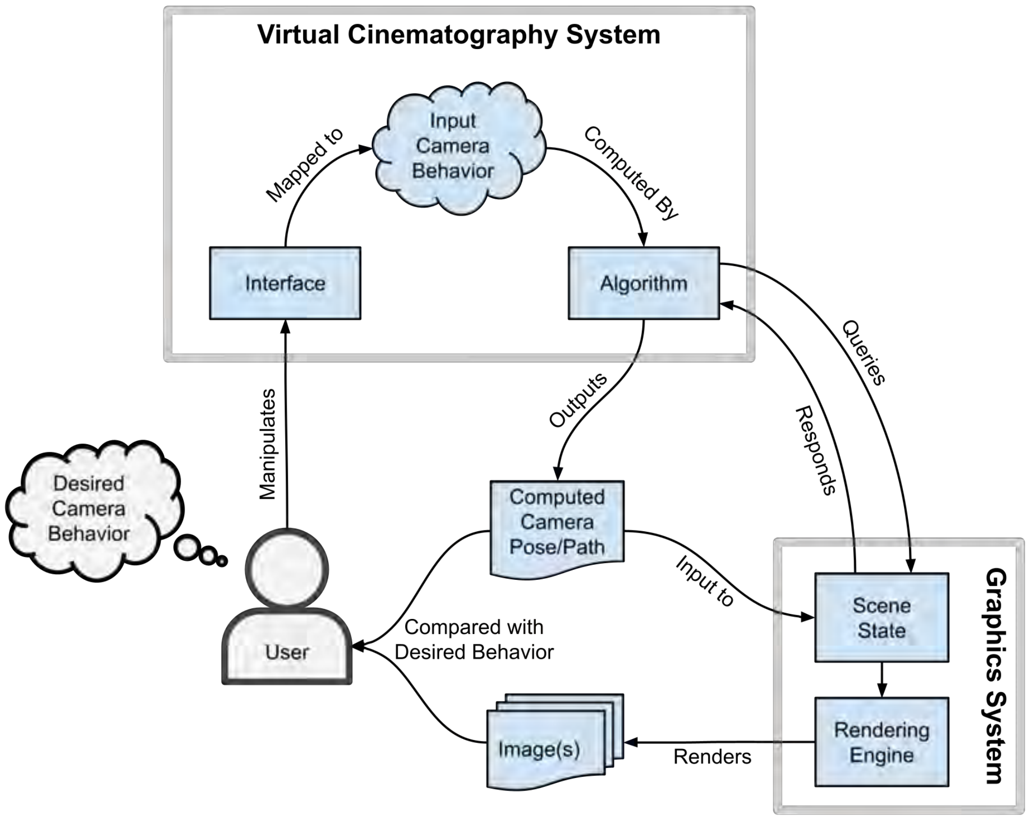


Figure 3.1: A high-level diagram of the structure of vector-based virtual cinematography systems.

1. Behavior types: What camera behavior does the user want? What range of camera behaviors can a virtual cinematography system produce?

2. Interface paradigms: How can a desired camera behavior be specified to a particular system? What types of behaviors are expressible with a particular interface?

3. Algorithmic strategies: How does a system compute a camera parameterization based on the specified behavior and available scene data? Does the user require more speed, efficiency, or breadth of computable behaviors than a particular algorithm is capable?

This behavior-interface-algorithm analysis strategy has several distinct advantages. Individually inspecting the supported behaviors, interfaces, and algorithms of systems dramatically increases the visibility of the similarities and differences between approaches.

Additionally, the types of behaviors, interfaces, and algorithms supported by each system can be shown to be directly and intuitively related to their expressivity, controllability, and efficiency.

Moreover, some types of systems can be understood as containing one or more sub-systems, each with its own set of behaviors, interfaces, and algorithms. By examining how higher-level desired camera behaviors are mapped into alternative behaviors solved by sub-systems' algorithms, these composite system approaches can be unpacked into more easily understood conceptual constituent components.

3.1.1 Behavior Types

How does the user want the camera to behave? Depending on user preference and application domain, any one of a vast array of desirable camera behaviors, spanning aesthetic, geometric, and functional considerations, may be desirable.

While the multitude of camera behaviors is far too large to be enumerated in detail here, many common camera behaviors share similar characteristics, and are analyzable through several different lenses.

One of the most intuitive and powerful analytical strategies examines each camera behavior as a combination of **behavioral properties** expected to be satisfied for the user to accept a candidate camera setting. At a high level, singular behavioral properties can be thought of as simple questions about the relationship between the camera, the computational state, and the object(s) of interest in the scene, such as the following.

- Visibility: can the object be seen in the image produced by this camera?
- Size: does the object appear to be a certain size in the image?
- Distance: is the camera within a particular distance range from the object?
- Frame position: is the object within a particular region of the frame?
- Relative Angle: is the object being viewed from a particular angle, relative to some

special direction?

- Occlusion avoidance: is anything else in the scene blocking the line of sight from the camera to the object?
- Collision avoidance: will the camera collide with anything in the environment as it moves?
- Smoothness: is the motion of the camera smooth?
- Path following: are some of the camera's settings following a specified path through the scene?
- Computational speed: can the camera settings be computed within a predefined window of real-world time?

While this list of simple behavioral properties is far from exhaustive, it is worth noting that many properties are closely related, with some intuitively translatable into combinations of others. For example, visibility can be thought of as a combination of frame position and occlusion avoidance, collision avoidance can often[29] be interpreted as a subtype of occlusion avoidance, and size can be thought of as either a sibling of the relative angle property, a subtype of distance in combination with a fixed field of view, or a pair of frame position properties. At the extreme of these close relationships, any and all of the properties above can be accomplished through a path following behavior, if the specified path is somehow chosen to match another collection of desired behavioral properties.

Additionally, these behavioral properties can be combined in a myriad of different ways, to which different systems supply differing support. Many systems, such as Blinn (1988), Ware and Osborne (1990), Shoemake (1992), and Lino and Christie (2012), only support **fixed** combinations of behavioral properties, allowing the user to modify some limited parameters without modifying the relative importance of any of the supported

[29]Assuming, as many software systems do, that the camera is an infinitely small point, that the unoccluded object is on the far side of collidable geometry from the camera, and that the collision geometry is identical to the occlusion geometry.

behavior's properties. However, a variety of systems have also been developed to support **arbitrary** combinations of behavioral properties. Some systems, such as Gleicher and Witkin (1992) and Christie et al. (2002), allow for dynamic sets of behavioral properties to be joined with simple logical conjunctions, with the user expecting all properties to be satisfied by the returned camera. Other systems, such as Ranon and Urli (2014) and Bares et al. (1998a), allow for different behavioral properties to be given different relative priorities, allowing for the system to automatically compromise if an ideal solution cannot be computed. Other systems still, such as Lino (2015), allow for complex logical and numerical combinations allowing for intricate balancing of behavioral properties.

A further distinction between virtual cinematography systems' behavioral combinations can be found in the number of objects over which the behavioral properties are permitted to be modeled. Many simple systems, such as Blinn (1988); Ware and Osborne (1990); Bares and Lester (1997); Zeleznik and Forsbergt (1999), are only capable of computing camera parameters relative to a single object in the scene, with all considered behavioral properties specified relative to that one object. While some other systems consider specialized behaviors over a singularly static number of objects (e.g., two objects (Lino and Christie, 2012)), other systems allow for behaviors to be specified over a variable number of objects. However, even this variability in trackable objects is not uniformly supported between different systems. Some systems allow for behaviors to be specified relative to a number of objects within a particular range (e.g., 1-3 Elson and Riedl (2007)), while other systems support any number of objects (Ranon and Urli, 2014; Lino, 2015).

Speaking broadly, some virtual cinematography systems are capable of large and diverse categories of behaviors, while others only support a few specialized forms. An intuitive metric for the expressivity of each virtual cinematography system might be to measure the size of the set of producible behaviors, although the close relationship between many groups of behaviors muddies the clarity of this notion somewhat. Should the production of a behavior that guarantees both frame position and occlusion avoidance count as a different from a behavior that guarantees visibility? How much does the desired angle from which an object is viewed need to change to count as a separate behavior? Worse still, different users may not be equally interested in the same regions of behavioral expressivity, or disagree on where the boundaries between specific behaviors might be, making any precise quantification of expressivity challenging. However, a simple qualitative comparison between two systems can be expressed by asking whether both systems are capable of producing camera instantiations that equivalently match a particular behavior.

3.1.1.1 Common Behavioral Specializations

The specialization of a group of behavioral properties can sometimes provide a direct practical benefit, allowing for greater efficiency in both communication from the user, as explored in Section 3.1.2, and computational speed. Although the realm of specialized behavioral properties explored in previous research is near boundless, the overwhelming majority can be understood as straightforward combinations of the simple properties above. Several specific combinations commonly appear in research literature and publicly released software.

For example, one natural and popular behavior has a camera smoothly follow a specified path through the scene, combining the smoothness and path following properties listed above. A user might input the desired camera pose at several explicit moments of time, and expect the virtual cinematography system to determine a path with some smoothness quality, usually mathematically formulated as differential continuity, that hits all those poses. Smooth path following behaviors are especially common in noninteractive applications, such as offline animation, where designers can precisely choreograph camera motion relative to the known motion of other objects in the scene long before display begins.

Within interactive applications, a common camera behaviors attempts to keep a set of objects visible at a set size in the frame, a category of behaviors sometimes referred to as *tracking* or follow cameras. Depending on how many visible objects are followed, their expected positions within the frame, and what camera properties are controlled, a few specialized forms of tracking camera behaviors are popular enough to have adopted unique terms.

One of the most ubiquitous tracking camera behaviors in interactive computer graphics is the *LookAt*[30] matrix behavior, which centers a given point in space within a camera's view by rotating the camera without modifying its position. Using the properties listed above, LookAt can be understood as a combination of frame position, with the object expected to stay centered on the screen, and path following, as the position of the camera is expected to follow a path provided by some external controller.

Many video games and other real-time applications use similar but more sophisticated camera behaviors, controlling the position, rotation, and sometimes field of view of the camera to make a single interactively controlled character appear at a specific distance and from a particular direction relative to the camera, a behavior commonly referred to as *third person view* (Haigh-Hutchinson, 2009). Correspondingly, *first person view* behaviors position the camera directly where a character in the game would be looking, for example from the point of view of a pilot in a cockpit or a soldier on a battlefield. The overwhelming majority of videogames in which the player controls a single character at a time employs either a first person or third person view camera behavior, with the parameters desired of the camera (e.g., distances, angles, and positions) specified by a combination of player input and game state logic.[31]

A slightly more powerful extension of this idea places several objects at fixed positions in the frame. One interesting example of this, that has appeared in several recent virtual cinematography publications such as Jiang et al. (2020) and Jiang et al. (2021), allows for two objects to be shown at specified positions in the frame and from particular angles.[32]

[30]Despite its ubiquity, the historical origins of the "LookAt" matrix are a bit mysterious. The first published appearance I have been able to find appears in the pioneering Blinn (1988), where it is called the "look-at" transformation. However, despite the lack of citations in the paper, Blinn's writing calls it a "traditional" technique, suggesting invention by an earlier party.

[31]Second person view, tracking the player's avatar from the perspective of another independently controlled character, is remarkably rare. I am currently aware of only one clear exemplar in a AAA game: the penultimate level of dri (2011).

[32]Note that these two behavioral properties in combination additionally capture the relative size of

As the region of camera positions from which two points can be positioned within the frame forms a torus, this is commonly referred to as a *toric space* behavior (Lino and Christie, 2015).[33] As users are often more interested in seeing one side of an object, like an actor's face or the front of a prop, much of that torus may be undesirable, requiring angular constraints to fully describe the desirability of a particular camera placement. As a result, this toric space behavior can be interpreted as a combination of multiple frame position and relative angle behavior properties.

3.1.1.2 Behavioral Abstraction and Style

This behavioral model implicitly assumes that the user already knows, and is prepared to precisely express, what behavior they want the virtual cinematography system to compute. However, this assumption does not fit all application domains and behavioral models equally well. For some users and application domains, the ability to abstract a complex combination of behavioral properties in to a simpler, more efficiently expressible format can be highly desirable.

Expressed in natural language terms, these behavioral abstractions are often of the form 'I want my camera to say X,' 'the camera should have Y style,' or 'I want the camera to mimic this video clip.' However, descriptions such as these can be profoundly ambiguous, motivating a variety of different approaches to mapping such ambiguous behavioral abstractions to concrete behavioral descriptions.

One simple form of behavioral abstraction mitigates the challenge of behavioral expression by allowing the user to choose from a small set of predefined behaviors. However, selecting a palette of predefined behaviors likely to be useful to a diverse body of users is far from easy. One such approach, appearing in works such as Christianson et al. (1996); He et al. (1996); Lino et al. (2010), and motivated by the work of film scholars

each object in the image, modeled as the angular size of each object as a sphere.

[33]Much of the mathematics of this method was first introduced in Lino and Christie (2012), although the phrase "toric space" did not appear until Lino and Christie (2015).

such as Arijon (1991), provides a library of predefined motion types corresponding to cinematographic *idioms*, including shot sizes and angles believed to be common. More recently, works such as Huang et al. (2021), and Ashtari et al. (2020), have indirectly presented the user with a handful of supported motion types, such as orbiting and following a given target. Some systems, such as Elson and Riedl (2007), take this a step further by asking the user to choose from a predefined library of keyframed camera motions, specified in a reference frame relative to the actors.

Another common behavioral abstraction is cinematographic style. Sometimes, these styles are assumed to be similarly finite and discrete, and are essentially interchangeable with the works described in the previous paragraph. However, some other systems consider style as a continuous space. For example, Jiang et al. (2020) and Jiang et al. (2021) map stylistic parameters, corresponding to how to mix together different types of motion, into concrete toric space behavioral properties. Additionally, these systems also support a style-by-example abstraction paradigm, where style parameters can be extracted from a video clip whose style the user wants to copy through automated computer vision analysis.

Across different domains, some users will come to the system undecided on what camera behavior is desirable, and intend to interactively explore the range of supported possibilities before coming to a final decision. While a particular camera behavior might be implicitly expected by the user in each moment of use, users in these highly interactive domains are commonly more conscious of how efficient and intuitive it is to make changes between closely related camera behaviors. While several concrete behavioral interface paradigms are discussed in Section 3.1.2, a number of abstract behavioral properties are often used to improve the controllability of these interactive interfaces.

One common behavioral expectation for interactive systems is partial changeability, where a specified portion of previously computed camera settings are left unchanged by future input changes. A particularly common instance of this is keyframe animation systems, in which the modification of a subset of keyframed camera parameters does not result in or require modifications to any other keyframes, allowing animators to iteratively improve the animation one moment at a time, such as explored in Jiang et al. (2021).

Some systems, such as Unity (2022), also support partial changeability of separable camera settings, such as automatically computing camera rotation changes to support modified behavioral properties while leaving the camera position unchanged.

3.1.1.3 Interpretation and Compromise

Suppose a user with a particular camera behavior in mind were to be presented with some candidate camera settings: how might they respond? What does it mean for camera settings to "match" a particular behavioral property? As with so many simple questions, multiple perspectives dictate diverging answers.

For some users and applications, individual behavioral properties represent requirements that a camera must either completely satisfy or entirely violate. Whether satisfactory camera instantiations are plentiful, singular, or nonexistent in a given scene, each candidate camera can be unambiguously evaluated as matching or not matching the behavioral properties. In this **rigid** binary model, the goal of a virtual cinematography system is to automatically identify a camera instantiation that matches the desired behavioral properties.

For example, the LookAt controller of Blinn (1988), arguably the most ubiquitous camera controller of all time, can be understood to compute a very rigid behavior. Given the expressed camera position and target position, the goal of the system is to compute a camera orientation that places the target exactly in the center of the camera's field of view. While some numerical error from floating point arithmetic may appear, if the system were to return camera settings that placed the target a noticeable distance from the center of the display then the user would find that their expected behavior is not satisfied.

However, such rigid interpretations can lead to a problem: what does the user expect to happen if no satisfactory camera instantiation can be automatically identified? This situation can easily and unanticipatedly arise if the user's behavioral requirements conflict with each other or the environment, a problem sometimes referred to as over-constrained specification. For example, a user might request that two objects be simultaneously visible in frame, a combination that may be impossible if the surrounding environment blocks the critical lines of sight. As another example, a user might desire a character be viewed from a particular angle and distance that would put the camera into collision with the environment. Worse still, the computation of camera settings matching a complex combination of behaviors might require more computational time than is allowable, such as during competitive video game play.

One approach to this sort of problem, employed in Christie et al. (2002) and Louarn et al. (2018) among others, is to simply report failure when detected, and request additional input from the user before computation is allowed to continue. For some user types and application domains, this rigidity may be desirable. For example, a user constructing offline scientific visualizations might find a candidate camera unacceptable if it fails to meet an exact set of standards, and welcome the opportunity to reevaluate what camera properties are necessary. Alternatively, offline animators making iterative modifications to character motions or environmental geometry might eagerly embrace the automatic reporting of any changes that would render their desired camera movement infeasible. However, reliably detecting if complex behaviors like visibility are truly infeasible, rather than simply difficult to compute, usually presents an immense computational challenge, as explored in Section 3.1.3.

Instead, many rigid systems solve this problem by automatically allowing some behavioral properties to be violated in order to completely satisfy others. For example, a virtual cinematography system struggling to find camera settings that make two objects visible in frame might choose to return a view in which only one object is fully visible, as Bares et al. (1998a) can. As another example, an algorithm might choose to diverge from the desired distance from which an object is viewed to show the object from a more important desired direction, if environmental collisions or occlusions render the ideal camera position infeasible, as Bares et al. (2000a) may. A strategy common in real-time graphics applications, such as in Lino (2015), is to return whatever camera settings can be computed in the time allowed, even if only some of the desired behavioral properties are satisfied, in order to have something to show on screen when required.

Fundamentally, approaches such as these attempt to automatically **compromise** between conflicting camera desires. Of course, a wide variety of compromise strategies are possible, ranging from maximizing the total number of properties satisfied as Burelli et al. (2008) does, to hierarchically relaxing lower priority requirements in deference to those of a higher priority as in Bares et al. (1998a).

Alternatively, some users and applications, such as Olivier et al. (1999), and Ranon and Urli (2014), expect support for a radically different behavioral interpretation style: by allowing for the expression of desired behavioral properties as measurably satisfiable preferences, a virtual cinematography system can instead be expected to compute camera settings that maximize the total amount of behavioral satisfaction, or equivalently minimize the total behavioral violation. While a measurably superior solution might be preferable, the user expecting their camera to exhibit some behavioral property in this style would be willing to accept any camera instantiation regardless of quality, if they are confident that no better possibility is available.

For example, rather than expecting a view of an object from a particular angle, the user might expect the camera to show an object from as close to a particular angle as possible, with small deviations from the desired angle preferable to large ones. Instead of computing a camera instantiation that fully satisfies a set of rigid behavioral requirements in a narrow computational time period, a common approach is to attempt to compute as good a camera as possible in the time available, as Ranon and Urli (2014) demonstrates. Another property commonly modeled this way is smoothness, with the camera expected to follow as smooth a path as possible while respecting a variety of other desired behavioral properties, such as in Lino and Christie (2015). However, balancing smoothness with other properties is a common source of difficulty in virtual cinematography, with some systems instead opting to discontinuously move the camera if continuing smooth motion is found to be unacceptably difficult (Giors, 2004; Galvane et al., 2013).

Nonetheless, sometimes the best available compromise solutions can produce undesirable results. For example, a camera expected to show as much of two objects simultaneously in frame might compromise by showing little to none of either, if the corresponding camera settings are the best found by the system. Depending upon the application, undesirable compromises such as these can produce serious consequences, from a videogame player losing a tournament match, to a medical professional missing evidence vital in the diagnosis of a patient's illness.

As a result, differing interpretations of how and when compromise from rigidity should be allowed represent a significant factor driving discrepant approaches to virtual cinematography. As computer scientists, most virtual cinematography researchers are accustomed, and often required, to think in terms of decidable formalisms that are commonly better suited to some behavioral interpretations than others, such as constraint satisfaction to more rigid interpretations, and optimization to more compromissory interpretations, among others. As such, the choice of behavioral interpretation is directly linked to the choice of formalism and, because disparate formalisms motivate divergent interface designs as well as algorithmic strategies, the general approach taken to virtual cinematography system design.

3.1.1.4 Behavioral Inputs and Outputs

Humans tend to think about and describe camera behaviors in terms relatable to natural language, for example "point at this object," "show me the front of this character," or "smoothly follow this path." Unfortunately, human descriptions such as these are often too imprecise or subjective to be expressible in a manner understandable or computable by a computer.

However, virtual cinematography systems are generally structured around more computationally concrete notions of behavior. In abstractly mechanical terms, each virtual cinematography software module can be thought of as taking inputs from the user and the scene, and outputting camera setting outputs following some computational procedure. The goal of every user is to efficiently find some configuration of inputs to a virtual

	Cam Pos.	Cam Rot.	Cam Lens	Obj Pos.	Obj Rot.	Environment
Visibility	Y	Y	?	Y	?	Y
Size	Y	Y	Y	Y	?	
Distance	Y			Y		
Central frame position	Y	Y		Y		
Frame position	Y	Y	?	Y		
Relative Angle	Y			Y	Y	
Occlusion avoidance	Y			Y		Y
Collision avoidance	Y					Y
Smoothness	Y	Y	?			
Path following	?	?	?			
Computational speed						

Table 3.1: Summary of which camera settings, object properties, and other environment geometry must be considered for each behavioral property in isolation. Cells with a 'Y' indicate that the column's setting must be considered for computation of the row's behavior, while an empty cell indicates that the setting is never required for that behavior. Cells containing '?' vary depending upon exactly what form the behavioral property takes. For example, computations concerning the frame position behavioral property will require cognizance of the camera lens settings if the desired position is not the center of the frame.

cinematography system that produces camera settings matching their desired behavior.

While the form of the data input and output may vary between specific interfaces and algorithms, the abstract content required to be considered represents a central characteristic of many types of behaviors.[34] The separation of that data between input and output in different virtual cinematography systems directly affects the manner in which behaviors can be compared, interacted with, and combined.

For example, consider two virtual cinematography system approaches, both from Blinn (1988), that attempt to position an object with a known position in the center of the camera's field of view. Abstractly, any system capable of automatically reasoning

[34]Different authors have suggested different ways of categorizing requisite data. For example, Christie et al. (2005) described a division between "on-camera" data, relating to the position and orientation of the camera, and "on-screen" data, relating to the properties of a visible object in the frame.

about the frame positioning of a particular must consider, at minimum, the position and orientation of the camera as well as the position of the object of interest in the scene.

1. One common and simple approach is the LookAt matrix, which centers the object in the frame by setting the orientation of the camera located at a given position. In terms of the behavioral property types described earlier, LookAt can be understood as a combination of frame positioning and positional path following behavioral properties. The LookAt matrix behavior can be interpreted as requiring, at minimum,[35] the position of the camera and the position of the object of interest as input, as no specific reasoning about the behavior can be accomplished without these properties. As output, the LookAt matrix behavior specifies rotational control for the camera that, if used with the given camera position, will situate the object of interest in the center of the frame. So, the inputs of LookAt must contain camera position and object position, and the output must contain camera rotation.

2. An equally valid, though less common, approach would be to position the camera with a given orientation so that the object is visible at a given distance in the center of the frame. In terms of the behavioral property types listed earlier in this chapter, this rotational approach might be understood as a combination of frame positioning, distance, and relative angle behavioral properties. In this behavioral approach, the inputs must contain the camera orientation, object position, and a distance from which to view the object, while the output must contain the camera position.

While both approaches can clearly accomplish the same frame positioning behavioral property, in which the object is centered in the camera's view, formalizing a comparison of the other behavioral properties in these differing approaches is less obvious. Examining the inputs and outputs of each approach provides a tidy answer to this problem: between inputs and outputs, both approaches consider the position and orientation of the camera as

[35]Usually, LookAt requires an additional directional input corresponding to the desired up direction which, together with the other inputs, fully specifies rotation in $SO(3)$. However, this technicality is safely ignorable for this abstract consideration.

well as the position of the object of interest, directly matching the minimum requirements for the frame positioning behavior. However, while LookAt takes camera position as input and outputs a camera orientation, the latter rotational approach takes orientation and distance as input to determine a camera position as output. Note that in both approaches to this behavior, all camera parameters other than position and orientation, such as lens settings like field of view or focus distance, can be freely modified without violating any of the stated behavioral properties, and are correspondingly absent from the inputs and outputs. Additionally, a keen reader might note the inputs to each system can be easily mathematically converted to the form of the other: this is no accident, as the underlying behavioral properties of both systems require cognizance of the same data, interpreted as corresponding to singular satisfactory camera solutions, regardless of the form the data takes.

However, a critical difference emerges when considering the ease with which these two systemic approaches might be combined with additional behavioral computations. For example, suppose the user wanted to directly specify a positional path for the camera to follow while keeping the object centered in the frame. Combining this with LookAt, which accepts a camera position as input, is considerably easier with LookAt than with the rotational approach. Alternatively, if the user wanted to view the object from a particular angle, perhaps to show one side of the object over another, incorporating this relative angle behavioral property would be substantially easier with a system already configured to take orientation as an input. This is fundamental to the combinational capabilities of systems like CineMachine, which use the separability of the desired behavioral properties to sequence algorithms capable of matching simple behavioral styles.

Analyzing the data captured in these inputs and outputs can also reveal what types of behavioral changes a particular system is capable of automating. For example, consider a camera that has its position and orientation directly specified by a user to point at an object of interest with a known position. Obviously, the camera is, by construction, matching a behavior of "position the object in the center of the screen," provided the object is always in the expected position. In this approach, the inputs and outputs to the virtual cinematography software module contains the position and orientation of the camera, but does *not* contain the position of the object. As a result, if the object's position is modified or in any way unpredictable, the resulting camera instantiation may no longer place the object in the center of the frame, violating the behavior. A system of this type can be described as incapable of automating the desired frame positioning behavior in the presence of variable object position.

3.1.2 Interface Paradigms

How should a user specify what they want the camera to do? While this question seems simple, the existence of any singular answer is remarkably ambiguous.

As already stated, users generally want to efficiently find a set or sequence of inputs for the system's interface that will cause the system to produce a camera instantiation matching their desired camera behavior. Additionally, each system may be capable of automatically producing a diverse multitude of distinct camera behaviors, few of which may correspond to the user's desired camera behavior. Providing an interface that allows for the efficient specification of a desired behavior while allowing as many behaviors as possible to be expressed, presents a daunting challenge, prompting a variety of approaches.

A common trend among many interface approaches is that supporting the expression of a greater amount or variety of camera behaviors often requires the user to provide more input data. As humans (and machines) generally take more time to input more data, increasing an interface's domain of expressible behaviors tends to come at the cost of reducing the efficiency with which it can be used. As a result, a quick at-a-glance comparison between interfaces can be made by considering a plot along two axes: the first corresponds to the number of automatable behaviors that are expressible in this interface (expressivity), while the second corresponds to the amount of information the user has to input to specify a specific camera behavior (verbosity). While the ease with which an interface can be controlled by a user (controllability) represents a complex mixture of objective and subjective considerations, systems that require less input do tend to be

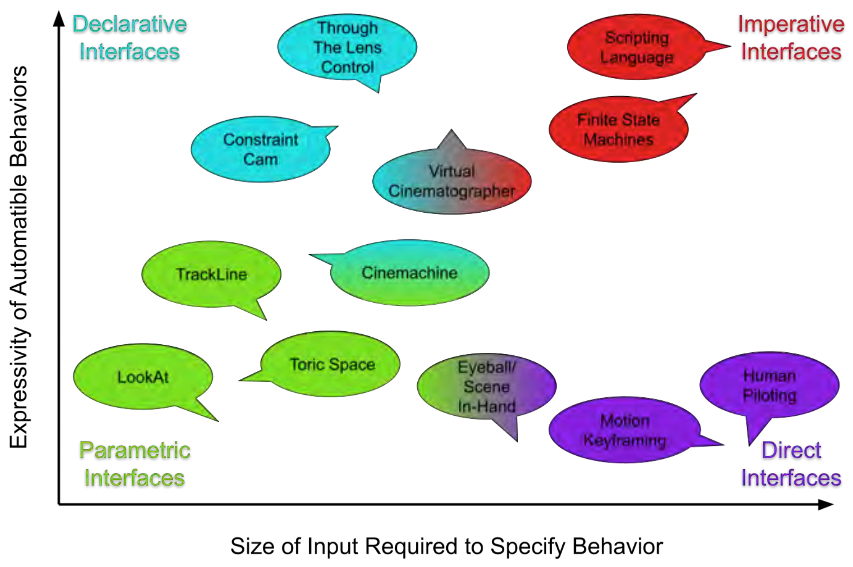


Figure 3.2: Diagram summarizing interface paradigms.

more controllable.

While a wide variety of virtual cinematography interfaces have been proposed in related research, this work identifies four categories of interface paradigms, referred to as **direct**, **parametric**, **imperative**, and **declarative** interfaces, which are discussed in detail in the following sections. These categories are meant to be neither exclusive nor exhaustive, but instead represent common interface modalities, with many different virtual cinematography systems exhibiting characteristics of multiple modalities. Additionally, the expressivity-verbosity plot can be roughly divided into quadrants corresponding to these common interface paradigms, into which virtual cinematography systems utilizing these paradigms tend to fall.

3.1.2.1 Direct Interfaces

Whether turning a crank on the side of a wooden box or inputting constants into a matrix in software, controlling a camera in the earliest days of filmmaking and computer graphics was an exceptionally manual endeavor. In some domains, modern camera systems remain reliant upon direct sources of input for the majority of their control. With these direct control interfaces, a signal from a human operator or camera-incognizant data source directly maps to camera system control inputs.

Computer animators and live action filmmakers frequently utilize direct control paradigms, with human artists and technicians directly piloting a virtual camera. Most modern 3D animation software, such as Maya, Cinema 4D, and Blender, by default require an animator to animate the camera in the same manner they would animate any other object in the scene, by directly specifying keyframed poses that may be interpolated between. For skilled offline animators, the directness of these interfaces can allow efficient expression of what may be highly complex behaviors. Additionally, by allowing a virtual camera to be directly puppeteered with a physical controller, such as in Ware and Osborne (1990) or other motion tracked controllers, users already skilled in live-action cinematography or other tasks can intuitively transfer their abilities to virtual cameras without having to learn any additional tools. However, if the primary objects of interest in the frame are significantly moved after the camera settings are input, a user will be required to manually modify the keyframes to achieve comparable behavioral properties.

Several videogame genres also commonly use direct control interfaces when the user needs and can be expected to have maximal control over the camera. Many flight simulators and vehicle racing games, for example, show the world from the pilot's point of view, with the camera effectively controlled by a combination of user vehicle input and software physics feedback. While such camera control is partially automated, the responsible software is not endeavoring to achieve any specific camera behavior, instead deriving camera parameters from algorithms designed for driving, flying, or some other camera incognizant process (Haigh-Hutchinson, 2009).

In the language of Section 3.1.1, direct interfaces are essentially only capable of expressing a single type of behavior: path following. Absent additional computational machinery to find a path matching other behavioral properties, direct interfaces represent the absolute minimum of expressivity over automatable behaviors. However, as already highlighted, direct interfaces can be highly desirable in applications where automatable expressivity is not required.

3.1.2.2 Parametric Interfaces

While frequently superficially dissimilar, a wide variety of virtual cinematography systems feature interfaces that share a common approach to user expression of camera behaviors: the user is presented with a fixed number of input parameters that are directly mappable to camera behaviors the virtual cinematography system is capable of automating. Regardless of how the input parameters are presented to the user, from manually modifiable text to interactively draggable image overlays, this document refers to this general strategy as the parametric interface paradigm. Broadly speaking, parametric interfaces provide a concise and controllable means of expressing a comparatively small family of related camera behaviors, on the assumption that the expressible behaviors will be valuable enough to the user to warrant specialized control.

Many of the earliest automated camera control systems support parametric interfaces, including the primary methods included in the pioneering virtual cinematography work, Blinn (1988). For example, the LookAt, described in depth in sections 3.1.1 and 3.1.3.3, can be understood as a parametric interface, where the inputs include camera position in $\mathbb{R}^3$, target location in $\mathbb{R}^3$, and an up direction in $\mathbb{R}^3$, for a total of 9 scalar inputs. While this LookAt interface maps to a single behavior, in which the camera position follows a path and a target object is centered in the frame, Blinn's paper describes a set of extensions to LookAt that allow for the position and eye target position to be replaced by other inputs representations transferable to LookAt, such as specifying a frame position for the target other than the center of the frame, and specifying a camera rotation but no position, among others. Despite their differences from LookAt, each of these alternative interface modes takes a fixed number of inputs, and all map to a small, closely related set of behaviors. For example, none of these interfaces are capable of expressing a camera behavior with more than one object in the frame, or a behavior that prevents occlusions blocking the visibility of target objects.

Another parametric interface is the toric space, described behaviorally in section 3.1.1, which allows for two objects to be shown at specified 2d positions in the frame and from particular angles, controlled with an interface with at least 4 scalar input parameters.[36] The originating paper of toric space control, Lino and Christie (2015), included several concrete user interfaces designed specifically to map to the positions of the two objects as well as the desired angles. While these toric space interfaces are undeniably valuable for scenes in which two humanoid characters were to be shown interacting on screen, any desirable camera behavior concerning a different number of objects/characters (e.g., 1 or 3+) is not representable with a toric space interface. Additionally, the expression of a behavior containing only a subset of the expressible parameters, for example a behavior in which the frame position of each object is specified, but relative angles are completely absent, is similarly poorly supported.

While most of the interfaces explored already provide a mapping from the user's inputs directly to behavioral parameters, other interfaces offer more abstracted parameterizations, as explored in behavioral terms in section 3.1.1.2. For example, Jiang et al. (2020) introduced a mixture-of-experts methodology to virtual cinematography, offering the user the option of mixing between several different specialized types of motion (e.g., track from the side, track while orbiting, dolly in/out) modeled as a recurrent neural network and based on an underlying toric space camera representation. By allowing the mixture parameters to be specified either manually or through the selection of an exemplar video clip, which is analyzed with computer vision to compute stylistically corresponding mixture parameters, their system allows for the efficient expression of subtly complex combinations of behaviors. However, because their method is based upon the toric space, it is limited to tracking exactly two subjects, with scenes containing more than two subjects only possible given additional explicit instructions as to when to switch between what pairs of subjects.

More examples of stylistic abstracted parametric interfaces? There are lots of drone

[36]The formulation in Lino and Christie (2015) requires FOV and an Euler triple for rotation relative to the axis between the two characters, for a total of 4 scalar parameters. However, the Euler rotation is derivable from frame position parameters given a known FOV. As a result, some alternative interface modes in their system supported different numbers or types of parameters corresponding to desired frame position, different rotation representations, or field of view, but each interface mode supports a fixed number of parameters directly mappable to the 4 standard parameters.

papers, what about pure virtual?

3.1.2.3 Imperative Interfaces

Imperative interfaces, sometimes alternatively referred to as procedural or forward control interfaces, are a common method for achieving almost complete automation: a user specifies desired camera behavior by providing a machine executable procedure to compute camera inputs. Because a user can adjust the computational efficiency of their camera system by tuning the system parameters or specifying a different procedure, imperative camera control interfaces are a popular choice for domains requiring high levels of efficiency, such as videogames and other real time graphics.

Most game engines, such as Epic Games (2022); Unity (2021), provide functionality facilitating user specification of imperative instructions, capable of camera control, as either plain text source code in a standard imperative programming language (e.g., C++, Python, C#) or through a visual scripting interface representing a standard programming language (e.g., Unreal Engine's Blueprints). Also arguably within the realm of imperative control, some game engines allow for camera control to be modeled with a user specified finite state machine (FSM). Several research systems have also been developed utilizing FSM architectures, such as in He et al. (1996); Burtnyk et al. (2002).

Modern AAA videogames regularly use incredibly sophisticated imperatively specified cameras, requiring scores of programmers and artists to implement and precisely tune the cameras to efficiently achieve desirable results. *Grand Theft Auto V* (2015) and *Red Dead Redemption II* (2018), to give two relatively recent examples, each credited 11 or more camera programmers or artists as part of their development (GTA, 2015; RDR, 2018).[37] Additionally, many of the details of these game camera systems are treated as proprietary,

[37]It should be noted that there is presently no widely accepted credits standardization in the videogame industry, either in terms of who should be credited or what titles should accompany various roles within development. As a result, it is likely that the total size of the team responsible for designing/engineering the camera systems in these games and many others is larger than credited (Moss, 2018).

with limited public discussion of any specifics.[38]

This begs a question: how does a user go about deriving a machine executable procedure matching their desired characteristics? Unless the characteristics in which the user is interested are very simple, specifying a matching procedure from scratch may be difficult for expert users and impossible for novices. A handful of systems, such as CINEMA introduced by Drucker and Galyean (1992), have attempted to provide collections of easily composable imperative primitives, although determining what composition is best suited to a desired behavior apparently proved troublesome to their users. A few game development textbooks and guides (e.g., Rogers (2014, p.131-161) and Haigh-Hutchinson (2009)) provide libraries of existing imperative algorithms for commonly in-demand types of camera behaviors. However, inconsistent labeling between sources can make locating an existing procedure that matches a specific camera behavior just as challenging as writing a procedure from scratch.

As a result, imperative interfaces feature an unsurpassed level of expressivity over automatable camera behaviors, but at a high cost to any camera designer hoping to specify a novel behavior type.

3.1.2.4 Declarative Interfaces

Declarative, sometimes called inverse, control systems provide what is arguably a more intuitive usage paradigm: a user specifies desired camera behavior by specifying what properties the resulting camera pose or path should have, leaving it entirely to the system to automatically identify a matching camera instantiation. Because declarative interfaces allow the user to specify arbitrary types and/or combinations of behavioral properties without having to explicitly provide instructions on how to compute a solution, declarative interfaces tend to offer a high degree of expressivity despite requiring a low amount of input verbosity.

[38]A notable exception to this rule is Giors (2004), which described the fundamentals of exactly how and why the camera system for *Full Spectrum Warrior*, a real-time-tactics shooter developed by Pandemic Studios, was developed.

However, a key question, intrinsically tied to the behavioral interpretation methodology discussed in section 3.1.1.3, must be asked of all declarative control paradigms: what specific form do the user's desired camera properties take? In formal mathematical terms, there are essentially three approaches to declaratively specifying a behavior: constraint satisfaction, unconstrained optimization, and constrained optimization, each of which is discussed in detail below.

While each of these formalisms have unique advantages and drawbacks, the fundamentally mathematical nature of these formalisms presents a controllability problem for users unable to express their desired behavioral properties in formal mathematical terms. To mitigate this issue, many systems instead support wrapping the underlying mathematical formalism in a more intuitive form, usually by allowing the user to mix predefined, parameterized behavioral properties with logical or numerical connection operators.

Unfortunately, there has been relatively little adoption of any extensibly abstracted declarative interfaces for virtual cinematography, resulting in most researchers introducing entirely new interfaces with each new system.

3.1.2.4.1 Constraint Satisfaction

In constraint satisfaction formulations, the desired camera properties are expressed as a set of **constraints**, each of which is either satisfied or violated by a particular possible camera instantiation. The goal of a constraint satisfaction based virtual cinematography system is to automatically identify a camera instantiation that satisfies all expressed constraints, if any such solutions exist.

Constraint satisfaction problems for virtual cinematography are usually expressed with scalar equality or inequality relations, for example

$$\mathbf{x} \text{ such that } c_1(\mathbf{x}) \geq 0, \ldots, c_n(\mathbf{x}) \geq 0.$$

This differs from other forms of constraint satisfaction problems common throughout traditional computer science, such as in artificial intelligence, that are usually expressed in the languages of first order logic and set theory.

Constraint satisfaction based declarative interfaces have been explored in a variety of sources. The Intent-Based Illustration System (IBIS), introduced by Seligmann and Feiner (1991), allowed the user to express their desired behavior as a prioritized conjunction of desired compositional properties (e.g., this object must be visible with the highest priority, this other object should be recognizable with a lower priority) through a text-based domain specific language. ConstraintCam (Bares and Lester, 1999) and its siblings (Bares et al., 1998b, 2000b) presented the user with a visual interface allowing for predefined constraint types (e.g., object size, relative angle, etc.) to be dragged-and-dropped on to the screen to form a conjunction of constraints.[39] The Image Descriptor Language (IDL), a text based domain specific language, allows for the expression of cinematographic behaviors (e.g., object size, visibility, etc.) in arbitrary logical combinations (i.e., mixture of conjunctions and disjunctions) Pickering and Olivier (2003); Pickering (2002). While its explored applications extend well beyond the boundaries of virtual cinematography, the text-based Cognitive Modeling Language (CML) has been shown to support expression of camera control as a constraint satisfaction based planning problem, with the user allowed to specify arbitrarily complex logic about what camera control types were allowed and what states were required to be achieved (Funge et al., 1999).

An important feature of constraint satisfaction problems is that all satisfactory instantiations are treated as being equally acceptable, with all unsatisfactory instantiations usually being equally unacceptable. A consequence of this feature is that, if no solution satisfying all constraints can be automatically found, a constraint satisfaction algorithm may return null (i.e., no valid camera settings available). This leads to the problem discussed in Section 3.1.1.3: what should happen if no fully satisfactory solution can be found?

Some approaches, such as Christie et al. (2002) and Christie and Normand (2005),

[39]While ConstraintCam could do many types of behaviors, it was limited, only capable of modeling behaviors with two or fewer on screen objects.

simply return that no satisfactory solution was found during a thorough[40] search, and expect the user to provide a set of modified constraints before continuing computation. In the realm of compromise interpretations, IBIS and ConstraintCam both support the specification of relative priorities of constraints, with lower priority constraints interpreted as being more violable than higher priority constraints (Seligmann and Feiner, 1991; Feiner and Seligmann, 1992; Bares et al., 1998a; Bares and Lester, 1999). IDL implicitly assumes that, in the event no fully satisfactory camera instantiation can be found, whatever instantiation satisfies the most constraints is most acceptable (Pickering and Olivier, 2003).

3.1.2.4.2 Optimization

The goal of an optimization based system is to automatically identify the best available instantiation of camera settings, with the user specifying a real valued **objective function** capable of evaluating the quality of each candidate camera instantiation. Note that maximizing an objective function for which higher valued outputs represent superior quality, is isomorphic with minimizing an objective function for which lower valued outputs represent superior quality, by simple additive inversion (i.e., the multiplication of the objective function by -1). Unconstrained optimization problems with objective function $f(\mathbf{x})$ are commonly mathematically denoted as

$$\arg\ \min_{\mathbf{x}} f(\mathbf{x}).$$

Unlike constraint satisfaction, unconstrained optimization treats all available solutions as acceptable, but endeavors to find a camera instantiation for which the objective function returns of as unsurpassable quality. As a result, an optimization algorithm will always return a non-null solution, even if all of the considered possibilities are of a relatively poor quality.[41] However, when the allowed computational time is limited, such as in real time

[40]Both systems use search algorithms that are slightly incomplete in practice. See Section 3.1.3.4.1.

[41]Ignoring the ever-annoying possibility of software errors producing NaN's for objective function

applications, the ability to accept the best computed camera solution allows for iterative algorithms to return a valid solution at any time.

A variety of systems have been developed that present the user with an unconstrained optimization based interface. The CamPlan system, introduced in Olivier et al. (1999), allows the user to specify a set of behavioral properties, chosen from a library of predefined objective functions, with the unweighted sum being optimized. Some systems, such as Burelli et al. (2008) and Litteneker and Terzopoulos (2017), additionally support weighting of individual properties to affect their relative desirability to the user. More complex behavioral property combination strategies in unconstrained optimization are relatively rare, but a notable example can be seen in Lino (2015), which proposed that simple mathematical functions (e.g., min, max, product) could be used to form arbitrarily complex combinations of predefined objective functions.

3.1.2.4.3 Constrained Optimization

Constraint satisfaction and optimization can also be combined into the form of constrained optimization, where the goal is to find the best solution that also satisfies all constraints, usually denoted as

$$\arg\min_{\mathbf{x}} f(\mathbf{x}) \text{ such that } c_1(\mathbf{x}) \geq 0, \ldots, c_n(\mathbf{x}) \geq 0.$$

While the ability to express both objective and constraint preferences provides a powerful boost of expressivity, general purpose algorithms for constrained optimization problems are far more difficult to implement, as explored in Section 3.1.3.4.3.

As a result, the full range of expressive possibilities of constrained optimization based declarative interfaces have been less widely explored. For example, Drucker and Zeltzer (1994), with the goal of automatically personalizing virtual museum tours, provides users the ability to select a set of constraints from a predefined list for the system to satisfy,

values.

as well as some controls over the parameters of an entirely predefined objective function. The Declarative Camera Control Language (DCCL) allows for desired behaviors to be specified as heuristics, in a Lisp-like domain specific language, corresponding to the quality of a transition between specialized camera controllers corresponding to cinematographic idioms, with the system expected to choose optimal transitions that do not violate predefined[42] continuity preserving constraints (Christianson et al., 1996). One highly influential constrained optimization system is Through-the-Lens Camera Control, which allows users to input both objective and constraint functions as functional expression graphs, although the form of constraints allowed is limited (Gleicher and Witkin, 1992).

3.1.2.5 Hybrid and Composite Interfaces

As previously mentioned, these direct, imperative, parametric, and declarative interface paradigms are intended to be neither exclusive nor exhaustive: a wide variety of virtual cinematography interface styles have been explored in related work, many of which can be interpreted as utilizing features of more than one of the paradigms listed here.

Many of these paradigmatic boundary crossers sprout from domains where users interactively explore 3D space in real-time. To give one prominent example, Ware and Osborne (1990) explored several different virtual cinematography interfaces that analytically mapped the 3D translational and rotational movements of a physical controller into direct camera inputs. These mappings were modeled around control metaphors through which the user is able to understand the relationship between the interface inputs and the camera motion by relating directly to familiar forms of locomotion. For example, the “eyeball-in-hand” metaphor provides a direct input to the camera, with the user’s 3D motions of the controller interpreted exactly as though they were holding the camera in their hand. A less direct, more parametric example is the “scene-in-hand” metaphor, which interprets user 3d controller motion as moving the entirety of the surrounding scene around a motionless camera. This “scene-in-hand” metaphor has proven influential across

[42]Their system theoretically supports extensible constraint specification, but the text suggests no scene or behavior specific constraint modifications were introduced throughout their experiments.

a range of interfaces for 3D scene exploration, with interfaces like Arcball (Shoemake, 1992) to work with 2D controllers such as a computer mouse.

Some interface styles, including the overwhelming majority of parametric interfaces, can be understood as providing a mapping from an intuitively controllable parametric space on to a subset of inputs presented by another type of interface. For example, the inputs to LookAt interface directly correspond to input parameters of an algorithm specifiable with an imperative interface, detailed in Section 3.1.3.3. Similarly, the toric space interface maps to specific factors of an optimization problem specifiable on a declarative interface. The style based mixture of experts systems explored by Jiang et al. (2020) takes this mapping a step further, by mapping user specified cinematographic style inputs to inputs for a machine learning derived imperative algorithm, which in turn outputs parameters that are mapped to the toric space parametric interface. More abstractly, all systems, regardless of interface type, can be understood to map from some user presented input space, through zero or more intermediary interface types, and finally to a direct interface of camera settings.

Some systems also offer the user with more than one type of interface. While relative rare in academic virtual cinematography projects, these composite, context-specific interfaces are common in software with wide application domains. One particularly prominent example, having achieved wider adoption among professional virtual cinematographers than its peers in recent years, is CineMachine, a virtual cinematography system packaged with the Unity game engine,[43] and intended to provide valuable camera control functionality in both online and offline applications. CineMachine achieves this by providing a variety of different interface styles corresponding to specialized camera control modules, such as algorithms designed to keep a character in a portion of the screen from a fixed distance. While several of these interfaces are essentially parametric, many modifiable either visually or textually, some of CineMachine's functionality is only accessible through

[43]CineMachine began as a start-up before being acquired by Unity Technologies in 2017. The CineMachine start-up was founded by Adam Myhill, a veteran game developer who joined Unity after the acquisition. Another major engineering contributor, sharing in several of the CineMachine patents, is Gregory Labute.

the specification of imperative extensions, principally to a module called the CineMachine "Brain" that is responsible for automatically switching between cameras, as C# source code (Unity, 2022).

3.1.3 Algorithmic Strategies

Algorithmic strategies for virtual cinematography are as diverse as the behaviors and interfaces they are designed to support. Some of the key differences between these algorithms were introduced in Section 1.1, including computational efficiency (e.g., real-time, offline), whether computation is online or offline with the display, and the amount of future knowledge the algorithm is provided. Additionally, the taxonomy of interfaces presented in the last chapter presents a valuable lens through which algorithmic approaches can be filtered.

- Direct interfaces: As direct interfaces represent the lack of meaningful automation, there are rarely meaningful algorithms employed for such interfaces.
- Parametric interfaces: As discussed in Section 3.1.2.5, parametric interfaces almost always map to another type of interface, usually specifying parts of inputs to imperative or declarative interfaces.
- Imperative interfaces: By definition, imperative interfaces require the user to specify machine executable instructions to produce camera settings matching their desired behavior. As a result, the algorithms packaged with the virtual cinematography system are often completely separated from the algorithms required for camera setting computation.
- Declarative interfaces: By shifting the responsibility for determining how to compute camera settings matching the user's desired behaviors from the user to the system, declarative interfaces place enormous demands on virtual cinematography system's algorithms. As a result, virtual cinematography research has explored a wide range of algorithms for constraint satisfaction, unconstrained optimization, and

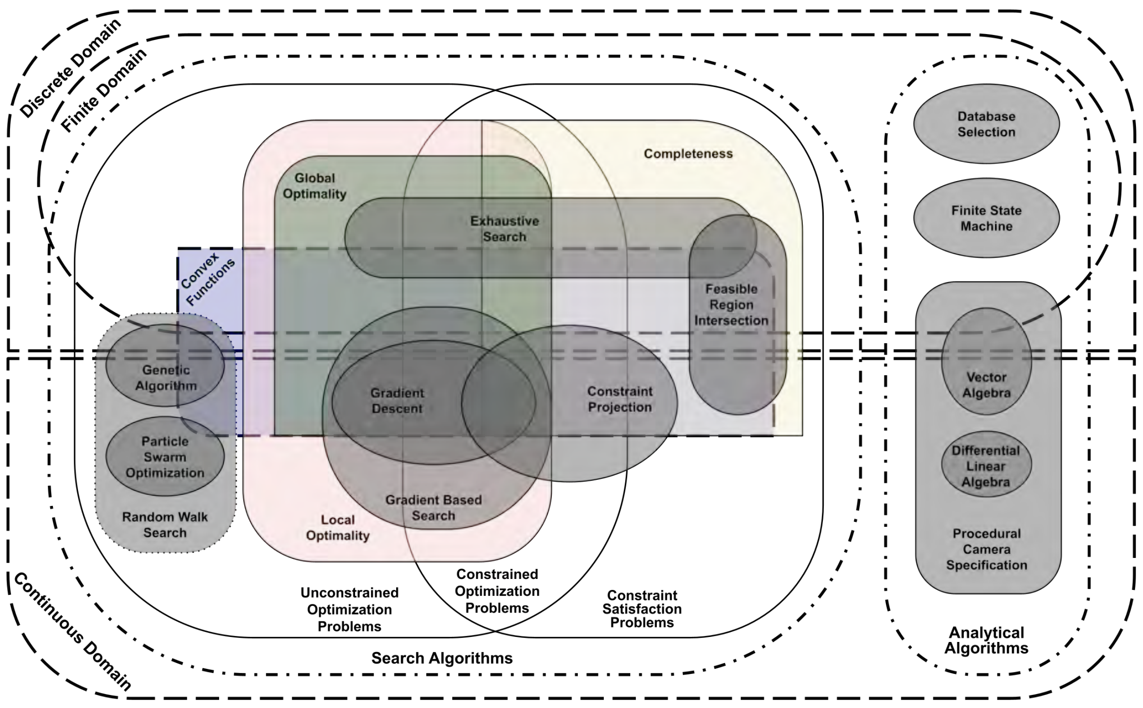


Figure 3.3: Euler diagram of virtual cinematography algorithms across different domains.

constrained optimization.

Consequently, this analysis can limit its focus to algorithms that stem from the use of imperative and declarative interfaces.

This work roughly categorizes algorithms for virtual cinematography into two groups. **Analytical** algorithms compute camera settings by following a fixed procedure, the result of which is guaranteed to be a solution matching the specified camera behavior. **Search** algorithms, by contrast, consider a volume of possible camera settings, and attempt to identify at least one solution that matches the specified camera behavior.

While a user could imperatively specify a search algorithm, most desired behaviors input with imperative interfaces are accomplished by specifying an analytical algorithm. Similarly, while some declarative behaviors can be solved by analytical algorithms, most declaratively specified virtual cinematography behaviors are only solvable with search algorithms.

3.1.3.1 Occlusion and Collision Testing

A challenge common to all algorithm types is that computing occlusion and collision cognizant camera instantiations is often difficult and expensive. As the camera's view can be occluded by, or its body collided with, any object in the environment, thorough occlusion or collision computation must consider all of the geometry in the scene, regardless of the level of complexity of the specified camera behavior. While collision testing is regularly computed with standard algorithms for computing intersections between geometric volumes, several different forms of occlusion testing algorithms have been explored in related virtual cinematography works.[44]

- Ray-casting algorithms, used widely throughout computer graphics including for Ray-tracing rendering techniques, identify one or more intersections with scene geometry along a line segment between a candidate camera position and a single target point. While a single ray-cast test is computationally inexpensive, many ray-cast calls, with a high cumulative computational cost, are sometimes required to determine the degree to which a particular voluminous object is occluded, or to locate a camera viewpoint in a search region with an unoccluded view of a target.
- Depth-buffering algorithms render the whole scene from a specific viewpoint to a raster pixel buffer, with each pixel identifying the nearest object in a particular direction as well as its distance. As a result, a single depth-buffering test executed from the viewpoint of a candidate camera can identify the degree to which a complex object of interest is occluded (Olivier et al., 1999). Alternatively, depth buffering from the viewpoint of an object can also be used to test whether candidate camera positions in a search region provide an unoccluded view (Burg et al., 2020). However, while hardware parallelization can significantly reduce computational costs, a single depth-buffering test is regularly more expensive than a small or moderate number of ray-casts.

[44]Christie et al. (2008) separated algorithmic approaches to occlusion as being either "reactive" or "deliberative," although the specific distinction between these classes was not thoroughly explained.

- Some systems instead attempt to directly compute what regions of the scene provide an unoccluded view of target objects, compiling the results into a data structure referred to as an aspect graph that can be queried with unparalleled efficiency (Plantinga and Dyer, 1990). However, accurately computing the aspect graph data structure in continuous space can be very expensive, potentially scaling to $\Theta(n^9)$ in the number of geometry primitives in the scene. Additionally, the aspect graph only remains valid if all occluding objects in the scene remain relatively static: if anything moves in a way that would change the topology of the aspect graph, the graph must be recomputed.

These methods vary in their applicability to different behavioral interpretations and scenes. Ray-casting algorithms cannot automatically identify whether any regions of the scene will have a fully unoccluded view of some target, but can trivially identify whether moving the camera a small distance in any direction will remove any occlusions, which is valuable for evaluating the optimality of any compromise solutions. Depth-buffering algorithms can approximately determine whether any regions of the scene provide unoccluded views, providing a useful mechanism for testing whether any rigid solutions exist, but have a much higher computational cost associated with determining whether any better solutions can be found in the neighborhood of the current candidate. While aspect graph tools are equally well suited to reasoning about both rigid and compromise behavioral interpretations, the cost of computation and the inability to move scene geometry severely limits applications compatible with aspect graphs.

Additionally, each method carries a wildly varying implementation cost. Virtual cinematography systems utilizing geometric collision testing or ray-casting occlusion testing tools are commonly implemented by borrowing geometry querying functionality from preexisting software, such as off-the-shelf game or physics engines, such as Epic Games (2022); Unity (2021), the vectorized operations or space partitioning data structures of which can significantly accelerate computation for complex scenes.[45] Similarly, depth-

[45]The techniques and algorithms necessary for efficient geometry querying are well outside the scope

buffering functionality is often achieved by integrating with existing real-time rendering technology, such as Kessenich et al. (2016). However, most researchers hoping to utilize aspect graph based queries seem to begin their implementations from scratch, possibly because there are fewer publicly available software packages directly suited to the task.

3.1.3.2 Temporality

Another distinguishing property between algorithmic strategies, both analytical and search based, rests in their approaches to camera settings across time, referred to here as **temporality**. Some applications only call for a camera pose for a single instant of time for each computation, while others require multiple camera poses corresponding to distinct keyframes, and others still require analytically defined curves to be output. There are three broad categories of algorithmic temporality common in related work.

- Many simple virtual cinematography systems, such as Blinn (1988); Pickering and Olivier (2003), compute camera parameters at each instant of time separately, with multiple **instantaneous** computations at distinct instants of time combinable into a camera path.

- Some other systems, such as Jiang et al. (2020); Huang et al. (2021), operate **sequentially**, with each computation producing camera settings for the current instant of time using inputs from the present and past states of the scene.

- Some further systems, such as Galvane et al. (2015a), output entire camera **path**(s) for a whole time period as the result of a single computation.

Each of these temporality approaches has potential advantages and drawbacks. Instantaneous algorithms can operate equally well regardless of keyframe count or density, but can face difficulty reasoning about behavioral properties like smoothness that require

of this text. An excellent overview of the mathematics of 3d intersection algorithms can be found in Schneider and Eberly (2002, ch.11), while many of the more efficient algorithms and data structures common in real-time software are detailed in Akenine-Möller et al. (2018, ch.22).

cognizance of other instants of time. Sequential algorithms are better suited to reasoning about smoothness, and are equally effective on paths of any length, but may face difficulty if keyframe density shifts, or if the user desires to manually edit part of the output path while expecting proceeding sections of the path unchanged. Path generating algorithms are often capable of reasoning about the most temporally complex behaviors, including smoothness, but often suffer from higher computational cost.

As with other categorical distinctions, these temporality categories are not exclusive, with many systems' algorithms employing multiple temporality strategies. For example, some systems output instantaneous or sequential camera settings for each keyframe in a long sequence, then post-process the path to enforce a desired level of smoothness before outputting the entire path (Jiang et al., 2020). A technique sometimes used with online sequential systems is to compute one or more paths into the near future, but only output the path's keyframe for the present instant of time, potentially unlocking for limited smoothness cognizance as well as occlusion/collision prediction in a sequential model (Funge et al., 1999).

3.1.3.3 Analytical Algorithms

There are many different types of analytical algorithms utilized in virtual cinematography. Many early systems, such as Chen et al. (1988); Mackinlay et al. (1990), follow simple vector algebra operations to compute a solution that solves a given mathematical equation.

LookAt from Blinn (1988), for example, takes as input a camera position $\mathbf{c}$, a target position $\mathbf{t}$ to be centered in the camera's view, and an up direction $\mathbf{u}$, all as column vectors in a homogeneous right-handed coordinate system. The LookAt behavior can then be expressed by first computing orthonormal bases $\mathbf{f} = \widehat{\mathbf{c} - \mathbf{t}}, \mathbf{r} = \widehat{\mathbf{u} \times \mathbf{f}}$, and $\mathbf{u}' = \widehat{\mathbf{f} \times \mathbf{r}}$, which can be used in matrix

$$M = \begin{pmatrix} I^{3\times3} & -\mathbf{c} \\ \mathbf{0}^\intercal & 1 \end{pmatrix} \begin{pmatrix} \mathbf{r} & \mathbf{u}' & \mathbf{f} & \mathbf{0} \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

This camera transform can be used to transform any point $\mathbf{p}$ from world space to camera space as $M\mathbf{p}$, with $M\mathbf{t} = \begin{pmatrix} 0 & 0 & -|\mathbf{t}-\mathbf{c}| & 1 \end{pmatrix}^\intercal$.

Other works have proposed different types of manually specified analytical algorithms, including finite state machines (He et al., 1996; Tomlinson et al., 2000; Burtnyk et al., 2002; Christie et al., 2002; Christie and Languénou, 2003), PID algorithms (Giors, 2004), and physics based algorithms (Turner et al., 1991). Some systems, such as Galvane et al. (2013); Unity (2022), analytically compute multiple camera paths simultaneously, choosing when and to what camera to switch based on separate logic. A variety of further manual analytical algorithm approaches are described in Haigh-Hutchinson (2009).

The rise of machine learning in recent years has prompted several analytical virtual cinematography algorithms specified with exemplar data. Some systems, such as those described in Chen and Carr (2015); Chen et al. (2016), attempt to learn a mapping from scene information and desired camera behavior directly to camera parameters by watching annotated video. Others, such as Jiang et al. (2020); Jovane et al. (2020); Bonatti et al. (2021); Jiang et al. (2021), have built systems designed to learn weights or parameters mapping to other imperative or declarative interfaces, such as toric space or motion style type.

Somewhere in between manually defined and machine learning algorithms are systems that attempt to match a user's desired camera behavior by automatically selecting a camera instantiation from a database. Cambot, for example, selects motions from ~ 50 predefined options based on the desired on-screen content specified with a domain specific language (Elson and Riedl, 2007). Some systems take this idea even further, facilitating the sequential combination of several different automatically selected predefined motions into a single smooth camera path (Sanokho et al., 2014). Database selection methods such as these can blur the line between analytical and search algorithms, at least as defined in this document. Under the hood, these database selection systems can be understood as seeking to identify an optimal camera instantiation, as described by a constrained optimization behavioral description, from a discrete domain with some type of, often exhaustive, search algorithm. However, all nontrivial database selection systems I have encountered require their search algorithms to be combined with considerable analytical computation to achieve acceptable results.

There are some explicitly declarative problems that can be completely or partially solved by analytical algorithms. For example, the toric space interface of Lino and Christie (2012) can be understood as reducible to a declarative problem specifying that two points $\mathbf{a}$ and $\mathbf{b}$ are expected to be separated by angle α from the camera's viewpoint, which is commonly analytically reduced to a surface of positional solutions on a torus centered on $\frac{\mathbf{a}+\mathbf{b}}{2}$, with an axis of revolution of $\widehat{\mathbf{a}-\mathbf{b}}$, a tube radius of $R = \frac{|\mathbf{a}-\mathbf{b}|}{2\sin(\alpha)}$, and a distance from the center of the torus to the center of the tube $R = rcos(\alpha)$. This torus can then be sampled parametrically with angular parameters θ and ϕ $\begin{pmatrix}(R + r\cos(\theta))\cos(\phi) & (R + r\cos(\theta))\sin(\phi) & r\sin(\theta)\end{pmatrix}^{\intercal}$ before being multiplied by a rotation matrix that aligns the model space z-axis with $\widehat{\mathbf{a}-\mathbf{b}}$, or combined with further geometric analysis to arrive at an even more efficient form. Some systems, such as Espiau et al. (1993); Marchand and Hager (1998); Marchand and Courty (2000); Courty and Marchand (2001); Marchand and Courty (2002) as well as a range of further works in the visual servoing field of robotics, present the user with a functionally declarative interface through which object points in 3d space can be constrained to particular regions in the image, and utilize a style of analytical iterative differential linear algebra algorithm for camera setting computation.[46]

Occlusion and collision avoidance are frequent areas of difficulty for analytical algorithms, with the direct computation of optimal camera paths that balance occlusion/-collision with other behavioral properties generally considered intractable for all but the simplest of scenes. Some analytical algorithms simply do not allow for occlusion or collision to be input as desired camera behavioral properties (Blinn, 1988). Among analytical algorithms capable of occlusion/collision cognizance, several divergent approaches to com-

[46]While these iterative numerical algorithms are similar to many of the optimization algorithms categorized as "Search" in this work, they are not general purpose: these numerical algorithms work for a very specific style of declarative constraint satisfaction problems, but cannot be applied to any other form of constraint satisfaction or optimization problem. It may be contentious, but I would argue that this prompts classification as analytical by the definition used here. Christie et al. (2008) classified these as a wholly different type of algorithm, which they termed "reactive."

promise have been explored. Some systems, such as Giors (2004) and Epic Games (2022, Default Third Person Camera), first compute an occlusion/collision incognizant camera instantiation, then use one or more raycasts to identify a nearby[47] unoccluded solution to what has already been computed. Alternatively, some systems, such as Seligmann and Feiner (1991); Feiner and Seligmann (1992); André et al. (1993), include whether to fade, cutaway, or omit occluding/colliding elements in the scene as part of the same pipeline that computes camera settings.

Regardless of the form of analytical algorithm chosen, the majority of analytical algorithms support a low number of virtual cinematography behaviors. While some analytical algorithms are capable of modifying or reprioritizing the behavioral properties to a limited extent, no analytical algorithm has yet been presented that is reliably capable of achieving arbitrary camera behaviors.

3.1.3.4 Search Algorithms

As discussed in Section 3.1.2.4, declarative interfaces require the user to specify how to test the quality of a particular candidate camera instantiation, but not a means to find one. The standard solution to solving declarative problems, whether they be constraint satisfaction, unconstrained optimization, or constrained optimization, is to utilize a search algorithm, which attempts to compute an acceptable camera instantiation by considering a range of possibilities.

Search algorithms take many forms, varying not only by what types of declarative problems are supported, but also by their computational speed and efficiency. One of the most critical criteria in comparing search algorithms is the size and structure of the space of solutions within which they are designed to search, with a stark division between **discrete** and **continuous** search domains.

[47] Getting more precise than 'nearby' reveals another set of differences between approaches. While Epic Games (2022) uses the intersection point of a single raycast to determine a 'nearby' unoccluded position, Giors (2004) can use a multiple raycasts along the prospective camera's left-right axis to identify subtle positional shifts that would maximize visibility, and further approaches exist.

Discrete domains represent a discontinuous set of possible camera settings. For example, a discrete domain might consider only moving the camera to one of the corners of a grid, but not between those corners. Most discrete domains in virtual cinematography are also functionally finite. If the size of such a discretely finite domain is small enough, it may be possible to **exhaustively** consider every possible solution, and pick the most optimal and/or satisfactory candidate. While a few published systems, such as Bares et al. (2000b); Vázquezz et al. (2001); Andujar et al. (2004), utilize such exhaustive search algorithms, most systems consider domains that are far too large to be exhaustively searched in a practical timescale.

Continuous domains represent camera settings that can vary continuously within some bounded or unbounded domain. For example, a continuous camera domain could allow a camera to move to any position within a large cube surrounding the scene geometry. As there are an infinite number[48] of possible choices in such a continuous space, exhaustively considering every candidate is impossible.

Large discrete domains and continuous domains therefore require search algorithms capable of automatically discerning which possibilities to devote computational consideration and which to skip. While these algorithms vary considerably between constraint satisfaction, unconstrained optimization, and constrained optimization, a common theme among all inexhaustive algorithms is that approaching absolute reliability commonly comes at the cost of enormous computational expense. Instead, a tradeoff is frequently struck between best meeting the user's specified declarative virtual cinematographic behavior, and the computing resources available before camera settings are needed by the user. As a result, inexhaustive search algorithms can be imprecise, incomplete, and unpredictable to differing degrees, but are the only currently known algorithms capable of supporting arbitrary camera behaviors.

[48]Or, if represented with floating point numbers, a number of genuinely astronomical scale. Current estimates place the number of stars in the observable universe at roughly $\sim 10^23$, and the number of atoms at $\sim 10^82$, while a 64 bit floating point number has $\sim 10^19$ possible values. A search with only two 64-bit numbers has more possible values to consider than there are stars in the sky, while a search with five 64-bit numbers has more possibilities than there are atoms in the universe.

Few, if any, of these search algorithms are unique to virtual cinematography, with many commonly used in other areas of computer science or applied mathematics. Optimization algorithms such as gradient descent and simulated annealing are common across not only computer science, but also economics, chemistry, and physics, among many other fields (Solomon, 2015). The constraint projection approach to constraint satisfaction problems is at the core of the physics simulation method known as Position Based Dynamics (PBD) (Weiss et al., 2018, 2017). Graph search methods such as A* have been widely popular throughout computer science for more than 50 years (Russell et al., 2010, p.109-111). This abundance of already available algorithms has meant few virtual cinematography researchers need to develop new algorithms for declarative behavior specifications.

3.1.3.4.1 Constraint Satisfaction

Algorithms used for solving constraint satisfaction problems are often classified as to whether they are **complete**, sometimes called systematic, where a complete algorithm is guaranteed to find a satisfactory solution if one exists. A critical corollary of this is that a complete algorithm asserting the absence of satisfactory solutions provides indisputable proof that no satisfactory solution exists. The set or volume of solutions that satisfy all constraints is commonly referred to as the **feasible region**, with any unsatisfiable constraint satisfaction problem having an empty feasible region.

Within discrete domains, complete search algorithms such as **A***[49] and other graph search algorithms popular in AI literature have been also been used for virtual cinematography (Funge et al., 1999; Oskam et al., 2009). If the constraint functions are convex, some systems, such as Halper et al. (2001), utilize a **constraint projection** algorithmic strategy, which iteratively projects candidate solutions on to the nearest point of any violated constraints until all constraints are satisfied, and is guaranteed to converge to a satisfactory solution if one exists.

[49]A* is effective with some flavors of constraint satisfaction and constrained optimization. See section 3.1.3.4.3 for more details.

Another complete family of algorithms, capable of operating in either continuous or discrete domains, works by taking the geometric intersections of the feasible regions of individual constraints, and is referred to in this text as **feasible region intersection** algorithms. Feasible region intersection algorithms are complete, as the intersection of any satisfiable set of constraints must be nonempty. With relatively simple (e.g., spheres or convex polyhedra with low face counts) feasible region geometry and small numbers of constraints, these feasible region intersection methods can be efficient enough to run in real time (Lino et al., 2010, 2011, 2013). However, employing similar algorithms for more complex geometry and larger sets of constraints is usually only tractable in offline computational applications (Louarn et al., 2020). However, a key prerequisite of all of these algorithms is the ability to efficiently compute the feasible region geometry of each individual constraint.

When these geometries cannot be easily computed from the constraint functions provided, alternative algorithms must be utilized. One such technique, utilized by Jardillier and Languénou (1998); Christie et al. (2002); Christie and Languénou (2003), identifies feasible regions by evaluating constraints across continuous intervals[50] of the search domain with tri-valued logical comparisons (i.e., outputs true, false, or could be true or false). Intervals which evaluate to true are fully satisfactory, and false are fully unsatisfactory, but intervals that may be true or false are subdivided into increasingly smaller segments to identify a geometric approximation of the feasible region. However, a limitation of this method is that greater numbers of subdivisions are needed as the feasible region grow smaller and more finely segmented, with the limit at an infinite number of subdivisions needed to identify a singular satisfactory solution in a non-singular domain. While this method is therefore incomplete, varying the depth of allowable subdivisions provides a mechanism for actively controlling the balance between completeness and efficiency.

[50]Note that this requires interval arithmetic evaluation of constraint functions, using similar rules to those described in Section 5.3.2 for simple closed form expressions. However, because these systems generally do not allow for constraint functions to have side effects or branching, they do not require the complex value range analysis tools proposed here.

Unfortunately, complete search algorithms are generally intractable for large sets of nonlinear, non-convex constraints (Russell et al., 2010, p.205-207). While a variety of incomplete, predominantly stochastic, constraint satisfaction search algorithms exist, I am not aware of any that have been applied to virtual cinematography thus far.

3.1.3.4.2 Unconstrained Optimization

Desirable unconstrained optimization solutions are traditionally classified as **globally** or **locally** optimal. While global optima must be at least as good as all elements in the entire, potentially infinite, search domain, local optima need only be at least as good as their immediate neighbors.[51] However, the ability to compute global optima often has less to do with the choice of algorithm than the properties of the objective function provided. Absent strong analytical guarantees about the objective function, such as linearity or convexity, determining whether a discovered local optimum is also a global optimum is effectively impossible (Solomon, 2015, p.166).

As few meaningful objective functions for virtual cinematography are linear or convex, even the most advanced optimization algorithms are frequently only able to guarantee local optimality for their returned results, although some algorithms may return global optimally results with nontrivial probability.

Perhaps the most ubiquitous optimization algorithm guaranteeing local optimality is **gradient descent**, which operates by iteratively translating a candidate solution in the direction that the objective function gradient identifies as 'downhill' until convergence.[52] As gradient descent requires many computations of the gradient, a key prerequisite is that the first order derivatives of the objective function must be efficiently computable, which often means that the objective function must be differentiable. A variety of further **gradient based optimization** algorithms have been employed to attempt to reduce the

[51] An input $\mathbf{x}$ is formally locally optimal minimum for a continuous function $f : \mathbb{R}^d \rightarrow \mathbb{R}$ if and only if $|\nabla f(\mathbf{x})| = 0$ and $\nabla^2 f(\mathbf{x})$ is a positive definite matrix, indicating that the curvature of f at $\mathbf{x}$ is positive in all directions. Within a discrete search space, local optimality is usually defined in relation to some explicitly defined neighborhood for each element of the space.

[52] Pseudocode for gradient descent is given in Algorithm 1 in Section 4.2.

number of computations required or otherwise improve computational efficiency, from Newton's method style algorithms which attempts to use the curvature (e.g., Hessian matrix) of the objective function to identify a more direct path to follow to reach the local optimum (Bonatti et al., 2020a,b), to Broyden–Fletcher–Goldfarb–Shanno (BFGS) style algorithms that attempt similar accelerations with curvature values estimated from analyzing changes to the gradient over multiple iterations (Yoo et al., 2021).

Some virtual cinematography systems employ algorithmic strategies that behave similarly to iterative gradient based optimization, but are not explicitly expressed as optimization problems. A family of **potential field techniques**, popular in robotics, operate by moving the camera in the direction of the gradient of a scalar field representing the desirability of camera poses throughout the scene (Xiao and Hubbold, 1998; Beckhaus, 2002). Others provide steering behaviors, borrowing from methods popular for flocking/crowd simulation, that directly define, for every possible pose in the scene, what direction the camera should move in this iteration (Galvane et al., 2013).

Alternatively, a variety of algorithms have been explored that have a practical probability of producing globally optimal results, but guarantee neither local nor global optimality. Such approaches are commonly referred to as **stochastic** search algorithms, as they commonly involve randomly sampling a probability distribution in which more probable choices correspond to more optimal solutions. For example, systems such as Olivier et al. (1999); Halper and Olivier (2000) utilize genetic algorithms, which use evolution inspired operations to iteratively improve the optimality of a population of candidate camera instantiations with selectional and combinational operations inspired by biological evolution. A stochastic algorithm particularly prevalent in more recent search based virtual cinematography systems, including Burelli et al. (2008); Burelli (2012); Abdullah et al. (2011), models optimization with motion rules inspired by social psychology over a population of candidate solutions, commonly referred to as particles, and is known as Particle Swarm Optimization (PSO).[53]

[53]Pseudocode for one type of PSO is given in Algorithm 4 in Section 4.2.

However, because stochastic algorithms are probabilistic, the probability of an objectionably suboptimal solution being produced may be unacceptably high. This probability can be tied to a number of factors, with larger populations and more algorithmic iterations often correlating with higher probabilities of success but slower performance. A key feature of many stochastic algorithms, including both genetic algorithms and PSO, is that they do not require derivative information, and can operate effectively with little to no information about the objective function being optimized. However, some systems utilize objective function information to increase the probability of success within a narrow computational time-frame. For example, Ranon and Urli (2014); Lino (2015) found significant performance improvements could be achieved by initializing PSO particles to optimize separate behavioral properties, as the interactions between particles allows for the best elements of each particle to be efficiently combined.

As mentioned in Section 3.1.2.4.2, a valuable property of all unconstrained optimization problems is that all solutions, even profoundly suboptimal ones, are valid. As a result, the intermediate results of all iterative search algorithms for unconstrained optimization can be used as camera settings regardless of whether the algorithm has converged or terminated, leading some researchers to term these as "anytime" algorithms.

3.1.3.4.3 Constrained Optimization

As mentioned in Section 3.1.2.4.3, constrained optimization problems generally combine the difficulties of constraint satisfaction and unconstrained optimization. Completeness, global/local optimality, and efficiency must be carefully balanced to achieve acceptable results on a practicable timescale.

One of the most common specialized forms of constrained optimization for virtual cinematography is commonly referred to as a path or **motion planning problem**, in which an optimal path is sought that will take the camera from an initial state to a target state using only allowed transitions between intermediary states (Funge et al., 1999; Oskam et al., 2009). Optimality for each path is evaluated relative to a sum of the cost of the transitions between subsequent states in the path, a property often referred to as distance.

When the state space is discrete, these motion planning problems are reducible to efficient graph search algorithms like A*, which can search large graphs by using a heuristically estimated[54] distance from a candidate next state to the target state, minimizing the number of states that must be considered. While such methods are guaranteed to be complete, the optimality and efficiency of the returned result is highly dependent upon the specific algorithm and heuristic chosen.[55] While algorithms for motion planning problems in continuous domains, such as rapidly expanding trees, have been developed in other fields, none have been employed specifically for virtual cinematography to my knowledge.

Inexhaustive search algorithms for more general forms of constrained optimization are relatively rare in virtual cinematography, with most systems utilizing some flavor of active set or interior point search algorithm. Such algorithms can be understood as variants of gradient based optimization that are combined with constraint projection to constrain search candidates at or beyond the constraint boundaries to the feasible region (Drucker, 1994; Drucker and Zeltzer, 1995; Huang et al., 2016; Bonatti et al., 2020a,b).

3.2 Separate but Related Fields

While attempting to understand the various approaches to virtual cinematography defined in the preceding pages, it can be valuable to understand what vector-based virtual cinematography is not. There are several bodies of related literature that clearly fail to meet one or more of the criteria for comparable systems listed at the beginning of the chapter, yet share significant strategies and techniques with immediately related work.

[54]Usually denoted as a function $h : n \to \mathbb{R}$, where n is a state or graph node.

[55]A* is guaranteed to find an optimal path if $h(n)$ does not overestimate the distance from n to the target. However, if $h(n)$ is constant for all n then A* essentially reduces to breadth-first-search, with exponential space complexity.

Researchers hoping to comprehensively understand virtual cinematography systems should at least have a passing familiarity with these separate-but-related problems, including an understanding of how these problems abut or overlap with vector-based virtual cinematography.

3.2.1 Robotics

The control of cameras in physical robotic systems represents an enormous body of research, and introduces a host of additional challenges (Chen and Carr, 2014).

Specific applications range from consumer electronics, such as controlling PTZ[56] or flying cameras to autonomously capture sporting events (Chen et al., 2016; Bucker et al., 2021), to surgical robots attempting to minimize the potential for patient injury while assisting human medical professionals (Hong et al., 1997; Chiou et al., 1998). Camera controllers in domains such as these play a more complicated and potentially dangerous game, using noisy sensor data to drive delayed actuators in the hope of achieving the desired imagery without inadvertently damaging the robot or anything in its environment.

At a high level, researchers in this field often, but not always, subdivide this challenging robotic camera control problem into three stages.

1. Sensor data is analyzed to understand the spatial relationship between the camera(s) and objects of interest in the scene. Where is the camera relative to the actors and environment?[57] Where are the obstacles to avoid? How are all of these objects moving relative to each other?

2. Next, the system must decide how to move the camera from where it currently is to a position that will provide the desired imagery using the robot's actuators without

[56]A standard type of robotic camera where motors allow the camera to pan vertically, tilt horizontally, and zoom in/out. This pan-tilt-zoom system is commonly abbreviated to PTZ.

[57]The problem of determining a robot's location from sensor data is commonly referred to as localization, and permeates a wide variety of robotics and computer vision literature. The task of **S**imultaneously **L**ocalizing the robot **A**nd **M**apping its environment is also closely related, and is usually abbreviated as SLAM. For the acronym obsessed, SLAM using only **V**isual sensors, such as cameras, is a task often abbreviated as VSLAM.

damaging anything. This is often expressed as a potential field or motion planning problem, which are discussed in sections 3.1.3.4.2 and 3.1.3.4.3 respectively. While the precise problem specifications and algorithms employed may diverge, this stage is fundamentally compatible with the virtual camera control problem central to this text.

3. Finally, the system determines what signals to send to the actuators (e.g., motors) as time progress, following the most recently computed motion plan. This is generally a dynamic decision making process that continuously incorporates sensor feedback to stably progress along the desired motion path, which is usually analogous to a closed loop control problem.

However, some robotic camera control systems have been built that utilize different processes. For example, several software systems designed for visual servoing tasks, where the motion of a robot is computed from visual sensor data, attempt to directly move from image features directly to motor inputs, which is sometimes referred to as visual servoing (Espiau et al., 1993; Marchand and Hager, 1998; Courty and Marchand, 2001).

3.2.2 Pixel-Based Approaches

As briefly mentioned in Section 1.3, there have been several pixel-based approaches to virtual cinematography.

Humans have been taking photographs and making motion pictures for many decades, and computer vision researchers have explored a myriad of techniques to identify aesthetic or communicative semantics in existing images. While this general strategy has found considerable success in image understanding, editing, and generation technology, using these types of tools to control how a camera moves through 3D space has so far born less fruit.

3.2.2.1 Inverse Rendering

Inverse rendering[58] asks a similar question to that of vector-based virtual cinematography: how should a computer choose a set of parameters, including scene geometric, lighting, and material settings as well as camera parameters, that will produce an image (or sequence of images) with desired pixel-based properties? In a sense, inverse rendering can be thought of as a super-problem of vector-based virtual cinematography: the control system can not only move the camera but also manipulate objects, materials, and lights in the scene, as well as evaluate the suitability of resulting images using the contents of their rasterized pixels.

There have been at least two excellent surveys on inverse rendering, Patow and Pueyo (2003) and Kato et al. (2020). In contemporary research,[59] inverse rendering problems are commonly expressed as optimization problems, with the desirability of a given frame computed by evaluating an objective function that takes rendered pixel data as input. This produces a problem form with an almost unbelievable level of expressivity, theoretically capable of everything from camera control to automatic lighting, and even 3D mesh reconstruction from a sketch (Poulin et al., 1997; Loubet et al., 2019).

While this is a much more powerful form of the problem, it is also far more difficult to solve in practice. Using almost any rendering method,[60] computational rasterization is generally discontinuous with respect to many scene parameters. With limited samples,

[58]Some recent authors, such as Kato et al. (2020), use the terms "inverse rendering" and "differentiable rendering" almost synonymously. I find this overloading needlessly confusing, so this document treats these as separate tasks with separate terms.

[59]The field has undergone considerable evolution in the years between these two surveys. One clear difference between these two surveys is that the later Kato et al. (2020) interprets everything as an optimization problem, while the earlier Patow and Pueyo (2003) lists many active (at the time) research areas that solve for parameters directly.

[60]Including ray/path tracing, fragment-based z-buffering, and neural rendering techniques.

continuous modifications to geometry,[61] lighting,[62] or material[63] properties can potentially produce discontinuous changes to individual pixels. As a result, optimization with pixel-based algorithms is profoundly challenging.

Up until a few years ago, there were two basic approaches to surmounting this challenge: either restrict the domain of allowed input to analytically solvable optimization problems, or use some flavor of search algorithm to numerically approximate an optimum. Unfortunately, suitable restrictions make the former hard to generalize, while the latter often carries a high computational cost. Many of the most efficient numerical optimization algorithms (e.g., gradient descent and its many variations) are reliant upon an approximation of the objective function gradient, which can be extraordinarily expensive to compute numerically (e.g., as finite differences) if the cost of rendering a single image is non-trivial.

While there have recently been significant advances in differentiable rendering[64], computing derivatives for gradient based optimization strategies remains significantly cheaper and more popular for vector-based optimization problems than it is for their pixel-based counterparts. However, differentiable rendering is currently undergoing a period of remarkable research progress (Li et al., 2018; Jakob et al., 2022), and the incorporation of this new generation of tools may shortly reduce the cost of pixel-based inverse rendering to a level practicable for commercial applications.

[61]As parts of geometry move, what pixels are occupied by this object obviously change. When the edge of an object intersects a pixel, the degree of overlap between the object and the pixel is almost always computed as an average of finite samples in the pixel, which produces discontinuous function behavior when the object's edge moves across a set of samples.

[62]While lighting intensity and color are generally continuous, the location or geometry of the light source are inherently discontinuous properties when shadows are considered.

[63]While most material properties, like color, are continuous and well-behaved, any material property that can modify the direction that light may travel after interacting with this material, such as roughness, may also produce discontinuities with global illumination.

[64]"Differentiable rendering" is itself an overloaded term in computer graphics. It is used here to refer to any rasterization process that can compute derivatives of pixel values relative to changes in geometry/material parameters. The same phrase is sometimes alternatively used to refer to multi-output rendering processes that are intended to be composited together as a post-processing step, for example by taking the weighted sum or difference of multiple layers.

3.2.2.2 Image Generation and Editing

One of the most popular areas of research in the computer vision community recently has revolved around machine learning techniques for the automatic generation and editing of images. Broadly speaking, the goal of these approaches is to output a grid of pixels that is as difficult as possible to distinguish from others in a given training dataset.

The presently dominant image generation strategy is to use a Generative Adversarial Network, commonly abbreviated to GAN, a machine learning architecture composed of generator and discriminator models. For image generation, the goal is to train the generator model to transform some input, often Gaussian noise, into an output image that the discriminator model cannot distinguish from images in the training dataset. With careful balancing of the generator and discriminator training regimes (Goodfellow et al., 2017, p.690-693), the resulting GAN can generate believable images, as well as any other type of data that can be fit into a static vector, including everything from furniture to aircraft voxel geometry (Li et al., 2017). However, research in this field is ongoing, with novel approaches producing unprecedented results constantly being published.

There is a greater diversity of strategies for image editing, with some divergence arising from the types of edits allowed. Common image recoloring tasks, for example, allow for properties, such as saturation and contrast, of the color distribution over the entire image to be modified to improve the aesthetic quality of the image (Koyama et al., 2017).

Additionally, some applications blur the boundary between image generation and editing. For example, some tools attempt to remove an undesirable element in the image and fill the resulting gap with believable content (inpainting) (Setlur et al., 2005), or make the image appear as though different light sources were present (relighting) (Sun et al., 2019). On the extreme end of this spectrum, novel-view synthesis tasks attempt to output an image from a novel viewpoint in a scene captured in one or more input images (Shuai et al., 2022).

It is worth clearly noting[65] that these sorts of image generation or editing tasks cannot be considered synonymous or generally interchangeable with the virtual cinematography problem formulation used in this work. With limited exceptions, the goal of image generation and editing tasks is to produce an image with desired properties, not to move or instantiate a camera within a scene.

One such exception lies in cropping tasks, in which a contiguous (e.g., rectangle or trapezoid corresponding to a homography) subset of pixel data is automatically selected and extracted into an output image with desired properties. For example, a user might want to remove an aesthetically unappealing element from their composition (Fang et al., 2014), or change the framing to shift the compositional emphasis towards an area of importance (Setlur et al., 2005). While the specific algorithms may vary, cropping tasks such as these can be interpreted as attempts to control the rotation and zoom settings of a stationary camera (Ronfard and de Verdière, 2022). A few groups have even implemented systems designed to provide smooth cropping control to attempt to mimic the behavior of a human camera operator (Chen and Carr, 2015; Gaddam et al., 2015), with some even going as far as to market their systems commercially (Technologies, 2022).

3.2.3 Computational Editing

A close sibling field of virtual cinematography is computational editing, for which an excellent[66] survey can be found in Ronfard (2012). Instead of attempting to select parameters for a camera, computational editing problems attempt to identify when and to what alternate view to discontinuously cut to achieve some desired communicative intent. In some scenarios, this can be understood as virtual cinematography across a discrete domain, with a single camera chosen from among several available to be shown at each moment. However, a critical difference is that computational editing software usually has

[65]If for no other reason than to attempt to prevent ML specializing computer vision researchers from asking why I would choose not to solve all of my virtual cinematography problems with a GAN.

[66]Albeit slightly out of date. Neither virtual cinematography nor computational editing had a new survey paper in at least a decade.

no influence over the camera views between which it is editing, instead deciding how to edit after the camera recording has completed.

Many early computational editing systems utilized either procedural, where a particular camera was to be used if some boolean condition was met (He et al., 1996), or logically declarative, where a sequence of shots that fully satisfies a set of binary rules is desired (Sack and Davis, 1994), approaches. However, neither of these approaches has proven particularly successful, as the former often requires an onerous amount of imperative input from the user, while the latter frequently provides too many satisfactory solutions to the user.

More recent approaches have expressed computational editing as what can be understood to be a combinatoric optimization problem, with each possible sequence of frames having a score for which the optimum is desired. In many systems, these scores are defined relative to idiomatically derived editing patterns or rules, such as to avoid jump cuts, to preserve screen directionality, and to show the most important characters at significant moments (Christie et al., 2012). Others have attempted to use machine learning (or data science) to have the computational editing software automatically construct editing patterns from existing edited videos (Merabti et al., 2015).

Solving the resulting optimization problem can be nontrivial, as the number of possible continuity preserving edits between M cameras across N edit points (e.g., moments when an edit can be made, or frames) is M^N. However, if the objective function is specified sequentially, with the choice of each edit point only influencing the score relative to the segments immediate before and after it, then the problem can be dramatically simplified into a form resembling a Hidden Markov model, solvable by dynamic programming (Galvane et al., 2015b) or the Viterbi algorithm (Leake et al., 2017).

Part of what makes computational editing so difficult is that the semantics of editing are poorly understood, as discussed in 2.4. There are often many valid ways to edit footage into a cohesive final film, with different editors and viewers frequently disagreeing about the quality and meaning of each edit (Lino et al., 2014).

As a result, few systems have proven capable of autonomously and reliably producing edits rated highly by diverse viewers. Instead, several systems attempt to provide partial editing automation to a human user, with a popular approach providing the editor with an automatically generated assortment of editing decisions alongside their traditionally familiar manual tools (Chen et al., 2013; Wu et al., 2018).

3.2.4 Character Animation

An enormous amount of research in computer graphics has been devoted to improving the efficiency, automation, and expressiveness of 3D character animation, as applied to everything from familiar human figures to the most alien creatures imaginable. Camera control and character animation are often treated as distinct tasks in practical computer animation and videogame development.[67]

However, some groups have attempted to build systems that bridge the gap between character animation and virtual cinematography, which can be appealing as both aspects of the animation are intrinsically related to the communicative content of the resulting images.

Some works, such as Elson and Riedl (2007), have attempted to simultaneously provide automatic control for both character animation and camera settings based on scripted narrative events to be displayed. Others, such as Chaudhuri et al. (2007), have developed controls that allow for efficient manipulation of character motion from the camera's viewpoint(s). Alternatively, some interactive systems internally represent the camera as a non-player-character (NPC) controlled by similar AI to other NPCs with which the player is interacting (Galvane et al., 2013), or as a directly linked component of another character in the scene. For example, the default third-person camera in Epic Games (2022) is created as a component of the tracked character, and connected by simulated springs.

[67]See Section 2.2 for a brief overview of why this is the case.

3.2.5 Automatic Layout Synthesis

Another significant sibling area of research concerns automatic layout synthesis, which generally seeks to automatically place one or more objects in a scene so that the final arrangement has some desired property. If one or more cameras are considered as objects to be placed, this problem can be interpreted as a super-problem of virtual cinematography, with both cameras and objects in the scene automatically controlled.

A number of early researchers in declarative computer graphics were motivated by automating the creation of technical illustrations, requiring the automatic selection and layout of specific components on complicated devices to be computed relative to user specified communicative rules (Seligmann and Feiner, 1991; Feiner and Seligmann, 1992; André et al., 1993; Butz, 1997). However, these early works were limited to relatively narrow types and configurations of communicative rules, requiring a high computational cost relative to the content produced.

While more recent works have dramatically improved the expressiveness (Yu et al., 2011) and computational efficiency (Weiss et al., 2018) of automatic layout synthesis systems, few have been developed that explicitly incorporate rules for camera control as part of the larger layout problem.

A notable exception is the work of Louarn et al. (2018, 2020), which uses the ability to control both camera and scene elements to automatically produce scenes that will produce desired cinematographic compositions. However, this massive increase in cinematographic expressiveness comes at a high computational cost, with complex scenes requiring over an hour of computation.

3.2.6 Computational Film Directing

Perhaps the most ambitious separate but related area to virtual cinematography, at least that I have encountered, lies in the realms of computational film direction. Many of the other areas covered in this chapter so far are directly analogous to skilled disciplines common in modern filmmaking, such as cinematography, editing, and character animation. For the majority of film productions, artists and craftspeople skilled in these disciplines must work in concert to make the intended story into a real film, with the entire enterprise taking its direction from the aptly named "director."

In an ideal world, computational film directors could produce complete, novel films wholly autonomously, and tailor each production to the viewers' individual preferences. However, as described in previous sections, the various disciplines that a computational film director must consider are each individually difficult, with any attempt at simultaneous control over the whole representing an incredibly ambitious task.

While many of the necessary components of a computational film director have already been discussed, several critically important consideration have not. Firstly, how should a story script be chosen? Secondly, how should a computational film director choose what camera behavior, character animation, scene layout, etc. to evocatively convey each moment of a scripted story? As with many other problems in this chapter, these are difficult problems to answer objectively, with many directions in which answers may be sought.

The idea of computational film directors is certainly not new, but it has only quite recently begun to be thought of as achievable, with Ronfard (2021) having published what is, to my knowledge, the first survey on the field.

3.3 Systems Engineering

It is worth briefly touching on the history of systems engineering in virtual cinematography. As detailed in Chapter 5, the software implementation of this work primarily takes the form of a novel programming language packaged with program transformation and static analysis tools. It is worth briefly discussing how this approach relates to the systems engineering of other virtual cinematography systems.

Unfortunately, virtual cinematography research has historically suffered from a lack of replicability, with relatively few papers releasing complete code for their implementations.[68] While some publications concern proprietary or privileged technology worth protecting, publishing code for even the least private of projects is not common practice. Separately, some researchers, such as Ranon and Urli (2014), helpfully publish a link to their code that has become invalid in the years after publication.[69] While publishing complete code has become somewhat more common in recent years (e.g., (Jiang et al., 2021)), doing so remains far from standard practice.

As a result, the practical engineering details of individual virtual cinematography systems are often unclear. However, reading the published prose descriptions of systems can reveal some common general engineering strategies.

Some systems implement singular behaviors with specialized, monolithic, and inextensible code that can only accomplish that singular behavior (Blinn, 1988). Some other systems utilize modular imperative implementations that are extensible by specifying more imperative code (Drucker and Galyean, 1992), with similar extensibility supported by finite state machine implementations (He et al., 1996). Some systems exploiting parameter separability or blending to combine several imperative control algorithms, which can be modified or added to by modifying the underlying source code (Unity, 2022).

In the realm of declarative virtual cinematography, Some declarative systems utilize the search functionality of preexisting computational systems like MATLAB (Huang et al., 2016), trading ease of development for dependency on the vagaries of a monolithic software system and potentially limited extensibility. Other declarative systems, such as Ranon and Urli (2014), have implemented new search functionality in modular software architectures that allow for new constraint/objective types to be added by modifying the underlying code base.

While effective, a drawback common to all declarative virtual cinematography systems

[68]A problem shared with many other areas of computer graphics (Bonneel et al., 2020), as well as a tragically high number of other fields of computer science.

[69]Ranon has since released an updated version of this codebase that can, at the time of writing, be accessed by the following URL: `https://github.com/robertoranon/Unity-ViewpointComputation`.

is that specifications of how to compute the value of an objective/constraint function are defined separately from how to compute derivative or range (e.g., interval) values necessary for some search techniques. This strongly differs from the methodology of my system, which incorporates automatic differentiation and range analysis techniques to automate these processes. To my knowledge, the work that comes closest to my approach is Gleicher (1994), which uses a graph representation of functional (i.e., side-effect free) expressions to allow for the automation of some differential linear algebra operations, including derivative calculation.

The specific algorithms and techniques my system uses to accomplish these tools are detailed in Section 5, which includes some brief discussions of the history of these works throughout computer science. However, to my knowledge, none of the specific techniques described in that section have previously been utilized for virtual cinematography.

3.4 Cinematographic Datasets

As discussed in Section 6, one of the contributions of this work is a novel dataset for vector-based virtual cinematography, constructed through automated analysis of feature films. While the idea of attempting to learn filmmaking through the analysis of films is hardly new, the construction of a dataset with sufficient size and granularity for generative filmmaking tasks is not straightforward. As collecting more data is expensive, researchers generally do not include more data in each dataset than is needed for their intended task, but exactly what is needed can vary dramatically depending upon the specific task.

As this text is primarily focused on vector-based virtual cinematography tasks, the primary data of interest pertains to the continuous motion of the camera and on-screen characters, specifically their position, orientation, and size in the frame, as observed in feature films. Unfortunately, few if any publicly available datasets have captured these properties at scale to date.

While film studios have used private records for financial and logistical purposes practically since their inception, the earliest work utilizing statistical analysis with respect to creative decision making was that of film scholar Barry Salt[70] (Salt, 1974), which spawned the long-running *Cinemetrics* film analysis project (Cin, 2022). Another prominent researcher with a similar approach is James E. Cutting, who has attempted to use film data to inform psychological research. While the available datasets contain human authored editing data (e.g., when each shot begins and ends) for tens of thousands of feature films, the few annotations regarding the contents of each shot (e.g., shot size, angle, etc.) are too coarse and small in number for vector-based virtual cinematography (Salt, 2006, 2009; Cutting et al., 2010, 2011).

A variety of further analytical filmmaking tasks have been inspired by natural language processing (NLP) concepts, including question answering (Lei et al., 2018, 2019; Tapaswi et al., 2015), summarization (Bain et al., 2020; Sun et al., 2022; Rohrbach et al., 2015), and scene segmentation (Rao et al., 2020b),[71] among others (Vicol et al., 2018; Gu et al., 2017). Broadly speaking, such tasks can be understood as seeking to teach computers to comprehend film story semantics using some combination of pixel data and annotations, including person bounding boxes, object/action labels, and written dialogue. While the publicly released datasets for these tasks are large in scale, with some such as Movienet (Huang et al., 2020) covering thousands of hours of films, and contain slightly more granular annotations, none I have examined contain data directly translatable to vector-based virtual cinematography.

By far and away, the dataset that comes closest to serving the needs of vector-based virtual cinematography is the *Film Annotation* dataset of Wu et al. (2017), which contains precise, manually supplied, annotations of the position, orientation, sizes of identified characters on screen. Unfortunately, most shots have only one frame of annotation, as the dataset contains only $\sim$ 1.2K annotated frames from $\sim$ 1K shots, which does not

[70]Salt has an unusual resume. Before beginning his work in film scholarship, he was a professional ballet dancer, computer programmer, and lighting cameraman, as well as completed a PhD in theoretical physics.

[71]Scene segmentation does not directly correspond to any common NLP task, but is often considered analogous to text segmentation: the division of a long sequence of symbols/frames into disparate, meaningful units, such as words, sentences, shots, scenes, etc.

significantly help characterize how the camera moves over time. While some datasets, such as MovieShots (Rao et al., 2020a), contain coarse annotations for more frames and shots, only *Film Annotation* includes continuous vector-based data.

Unfortunately, a frustratingly common practice among machine learning based virtual cinematography researchers, such as Jiang et al. (2020); Courant et al. (2021); Chen et al. (2013); Chen and Carr (2015); Chen et al. (2016), is to not publish their datasets. I can only speculate as to why this has come to be, but a common sticking point in the computer vision community revolves around legality, a topic about which I am admittedly inexpert.[72] To put it very briefly, most legal systems assign a notion of intellectual property ownership (e.g., copyright) to films similar to other creative works, such as books or music. Many types of use of a film, including copying, showing to an audience, selling merchandise based on the film, etc., require explicit advance permission of the owner of the film's intellectual property, usually the organization that produced the film. While home viewing of a legally owned copy of a film, as well as some types of use for scholarship and noncommercial research, is commonly considered to be legal, sharing digital copies of films between researchers without explicit permission from the films' owners is commonly viewed as comparable in illegality to piracy.

Some researchers, such as Galvane et al. (2015b) and Phillipson et al. (2022), have attempted to circumvent this limitation by only sharing data on films they have produced themselves, thereby allowing the researchers to assign whatever rules on copying/sharing they wish. However, the production of even the smallest scale film is still an expensive endeavor, significantly limiting the scope of capturable data. Another method for circumventing this issue is to instead publish annotations only on legally accessible film clips, such as trailers and scene excerpts published during films' marketing campaign, theoretically allowing researchers to legally share annotations for films their colleagues can legally access (Bain et al., 2020). However, a limitation of this approach is that the domain of the dataset only consists of clips selected by each film's marketing team, which

[72]To be absolutely clear, I am not a lawyer. Nothing in this entire document constitutes legal advice.

carries a significant risk of unwanted bias in the data. Additionally, the accessibility of the film clips is likely outside the control of the researchers involved, with the potential to disassociate the annotations from viewable video without warning. As a final fallback, some datasets simply publish annotations without providing any access to the underlying video.

Regardless of the source, data collection, or dissemination methods chosen, gaining access to a cinematographic dataset is a far cry from putting the dataset to practical use. Machine learning and data science for virtual cinematography have historically been relatively rare, with the most common methodology being to attempt to learn a simple parametric interface, usually mapped to one or more simple imperative procedures, that directly produces camera parameters. I am aware of no published work that has attempted to automatically learn any kind of declarative interface for virtual cinematography from data.

CHAPTER 4

Mathematical Framework

How should a user input their desired cinematographic preferences, and how should a system automatically identify a matching camera instantiation? While different researchers have employed a variety of system paradigms in answering these questions,[73] the approach I have pursued is to formulate the problem as a continuous unconstrained optimization problem, where the objective function is specified to have minima where the user's desired cinematographic behaviors are best satisfied. In this paradigm, a suitable camera instantiation can be formalized as

$$\arg\ \min_{\mathbf{x}} f(\mathbf{x})$$

given a user specified $f : \mathbb{R}^d \mapsto \mathbb{R}$ and a continuous domain to search within that is a subset of $\mathbb{R}^d$. To simplify notation, various functions are denoted as $f(\mathbf{x})$ throughout this document, despite potentially requiring different numbers and types of arguments.

Given that the intended target audience of the system include cinematographers, game designers, and other craftspeople whose skills may not extend to high-level mathematics, any usable system should provide an interface that abstracts away the most technically challenging or complex elements of the underlying optimization problem, instead accepting expressions of desired behaviors in terms with which the user is more familiar. However, deciding what mathematical tools the system should support requires careful consideration of some of the types of abstraction a user might desire:

- Set of behaviors: A user might desire a camera instantiation matching any number

[73]As discussed at length in 3.

of desired shot behaviors. For example, they might want a particular actor to be visible in the image, to have a particular size, and to be viewed from a particular angle. If each behavior is modeled as an independent objective function, how can they be combined to form a single objective function?

- Hierarchical behavior preference: A user might feel that some behaviors are vital while others are merely optional. Perhaps the user feels that the actor must be visible in the image, even if it means that they will have a different size or be viewed from a different angle. How can these hierarchies of behavior preference be expressed as an unconstrained optimization problem?
- Variable precision: Depending on application domain, the quality of optimum a user might find satisfactory may vary wildly. For domains such as video games, a user may want whatever optimum the system has found after a given time period. In domains such as offline animation, a user might want to see rough results quickly but better results when available, with the system stopping the search only after finding a result that meets or exceeds a specified threshold of quality. Are there types of objective functions or optimization algorithms that are better suited to these differing approaches to precision?
- Alternate use cases: A user may wish to use the same virtual cinematography system for different applications, with variations including whether the system has any knowledge of future knowledge while computing for each moment of time, and how much time is allowed before the computation is required to complete. How compatible are different objective functions and optimization algorithms with such a diversity of applications?

What kinds of mathematical tools are needed to support such abstractions?

4.1 Objective Functions

To find a camera pose, the system must be able to evaluate how well a particular camera pose matches the user's desired shot behaviors. As we are using an optimization based approach, this is done with an objective function $f : \mathbb{R}^N \mapsto \mathbb{R}$ representing the user's desired shot behaviors, with the desired camera instantiation within $\arg\min_{\mathbf{x}} f(\mathbf{x})$.

Within this work, any objective $f(\mathbf{x})$ is assumed to be an instantaneous[74] function, for which $\mathbf{x}$ represents the state of the camera at a single instant of time. Evaluating the objective function over an entire camera path requires the parameterization of $\mathbf{x}$ over time to $\mathbf{x}(t)$, which can be evaluated as $\int f(\mathbf{x}(t))dt$ over some time interval. To simplify computation, the system operates in discrete time, with the camera specified by some set of keyframes at times $T = t_1, \ldots, t_n$ allowing the objective function for a path to be evaluated as $\sum_{t \in T} f(\mathbf{x}(t))$.

However, as mentioned at the beginning of this chapter, users often want more than one behavior to be satisfied, with each behavior independently modeled as a separate objective function $f_1, \ldots, f_n$. One simple and popular method for combining multiple shot objectives into a single objective function is as a weighted sum of the these independent objective functions,

$$f(\mathbf{x}) = \sum_{k=1}^{n} \alpha_k f_k(\mathbf{x}).$$

Not only does this method allow the influence of each objective to be tuned by adjusting the values of α, but it also critically allows for compromises between shot objectives for which the minima do not precisely intersect.

[74]As opposed to a sequential or whole-path function. See Section 3.1.3.2 for a discussion of differing temporalities in related work.

4.1.1 Hierarchical Constraint Penalties

As discussed in Section 3.1.2.4.3, sometimes a user may want to specify their desired camera behaviors as

$$\begin{aligned} \arg\min_{\mathbf{x}} \quad & f(\mathbf{x}) \\ \text{s.t.} \quad & c_1(\mathbf{x}) \leq 0 \\ & \ldots \\ & c_n(\mathbf{x}) \leq 0. \end{aligned}$$

Aside from personal paradigmatic preference, a user may wish to use this constrained optimization format to explicitly express a partial ordering over their desired cinematographic behaviors, with higher priority preferences appearing in $c_i(\mathbf{x})$ and lower priority preferences appearing in $f(\mathbf{x})$. This provides an alternate method of controlling compromise behavior when all desired behaviors are not mutually achievable in a given scene, allowing a user to specify, for example, that one actor's visibility in the frame is critical while another's is simply optional.

However, without unreasonably strong guarantees on the behaviors of the constraints or objective function, such as linearity or convexity, implementing a system capable of solving a constrained optimization problem is fraught with difficulty. For example, what should the system do if the feasible regions of the constraints do not intersect? Will the user expect the system to exhaustively prove the unsatisfiability of the problem before reporting an error, or will the user be satisfied by a set of prospective minimums which satisfy as many of the constraints as possible? Any attempt at answering these questions requires assumptions about the nature of the application in which the system is being used, as well as the user's ability to understand relatively sophisticated mathematical tools.

An appealing alternative is to transform the constrained optimization problem into an unconstrained optimization problem for which any global minima must satisfy as many constraints as possible. Such techniques are commonly referred to as penalty methods,[75] with the usual form adding some penalty $g(\mathbf{x})$ to the objective function, so that $f(\mathbf{x}) + g(\mathbf{x})$ is minimized if and only if as many of $c_1(\mathbf{x}), \dots, c_n(\mathbf{x})$ as possible are satisfied. A traditional choice for a penalty function $g(\mathbf{x}) = k \sum_{i=1}^{n} \max(0, c_i(\mathbf{x}))^2$ for some tuned weight k.[76] While effective, this traditional approach fails to support any specification of relative preference between constraints, preventing any control over the how to compromise between constraints that are not mutually satisfiable.

To combat this problem, this work introduces a novel hierarchical penalty method formulation, which allows for a constrained optimization problem to be reduced to an unconstrained optimization problem with an added penalty corresponding to a partial ordering of constraint priority.

To construct this hierarchical penalty method, first take the case of a constrained optimization problem with an objective function $f(\mathbf{x})$ and constraints $c_1(\mathbf{x}) \le 0, \dots, c_n(\mathbf{x}) \le 0$, all of equal priority. Critically, if

$$\exists a, b \text{ such that } \forall \mathbf{x} \in \mathbb{D}, -\infty < a \le f(\mathbf{x}) \le b < \infty$$

within some search domain $\mathbb{D} \subseteq \mathbb{R}^d$, then the penalty function

$$g(\mathbf{x}) = (b - a + \epsilon) \sum_{i=1}^{n} h\left(k_i c_i(\mathbf{x})\right),$$

where $k_1, \dots, k_n$ are scalar weights, and $h(x) = \frac{1}{1+e^{-x}}$, a standard logistic function.[77] With this penalty function, it must be that $g(\mathbf{x}) > f(\mathbf{x})$ if any of the constraints are substantially unsatisfied at $\mathbf{x}$, guaranteeing that any optimization search will always prefer

[75] Some sources, such as Solomon (2015), refer to this idea as a "barrier" method rather than a "penalty" method, although both terms are somewhat overloaded throughout literature.

[76] Many penalty method optimization algorithms increase the value of k with each search iteration to force $g(\mathbf{x})$ to dominate any residual in $f(\mathbf{x})$.

[77] It is useful to note that $\frac{dh}{dx} = \frac{x' e^{x}}{(1+e^{x})^2} = x' h(x)(1 - h(x))$.

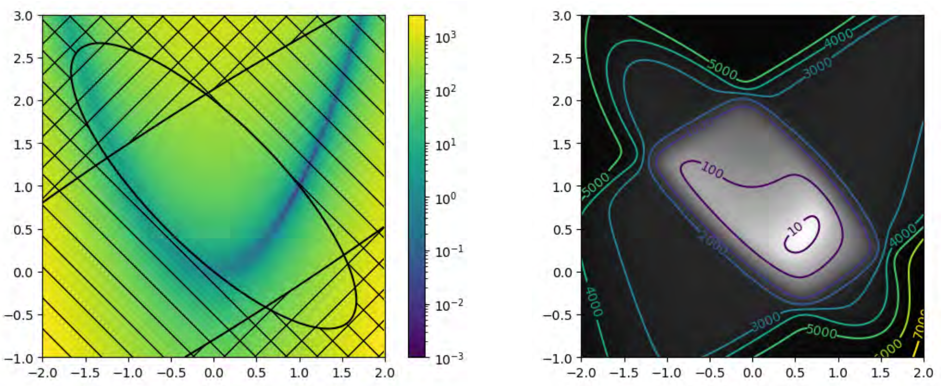


Figure 4.1: Example of a function with constraints, and a corresponding hierarchical penalty derived unconstrained version. On the left is a colored plot of an example objective function, the values of which are colored according to the color bar legend, with two constraints denoted with diagonal hatching. On the right is a colored plot with contours to show the result of combining the objective function with hierarchical penalty functions formed by the constraints.

solutions with fewer substantially unsatisfied constraints.

However, the constraints $c_1(\mathbf{x}), \ldots, c_n(\mathbf{x})$ are so far considered of equal priority. Suppose that there is an additional set of constraints, $b_1(\mathbf{x}) \leq 0, \ldots, b_m(\mathbf{x}) \leq 0$, that are to be considered as higher priority than $c_1(\mathbf{x}), \ldots, c_n(\mathbf{x})$. This can be simply accomplished using the tools already specified by considering $f(\mathbf{x}) + g(\mathbf{x})$, expressed over $c_1(\mathbf{x}), \ldots, c_n(\mathbf{x})$, to be an unconstrained objective function subject to constraints $b_1(\mathbf{x}), \ldots, b_m(\mathbf{x})$. By noting that,

$$\forall \mathbf{x} \in \mathbb{D}, a \leq f(\mathbf{x}) + g(\mathbf{x}) \leq n(b - a + \epsilon),$$

a new penalty $g'(\mathbf{x})$ over $b_1(\mathbf{x}), \ldots, b_m(\mathbf{x})$ of the same form can be written with new factors $a' = a$ and $b' = n(b - a + \epsilon)$.

This recursive stacking of constraints allows for any hierarchical preference of constraints to be reduced to an unconstrained optimization problem, allowing for any unconstrained optimization algorithm to be used. However, this hierarchical penalty method is not without complications.

Arguably the most significant complication is that any underlying objective function must have a finite range that is known. Given that part of the goal of this work is to develop systems capable of being used by a variety of different users, it is highly undesirable to require users to manually specify the range of their objective function through pencil-and-paper calculations or trial-and-error. Instead, the system developed includes functionality to automatically estimate the range of expressions, as described in Section 5.3.2. It is useful to note that the range bounds do not have to be particularly tight, although excessively large values can lead to numerical precision issues. However, some care with objective function specification to ensure that the actual range of the function is finite, with one useful technique for theoretically infinite functions being to clamp the function within a large but finite range.

An additional, but arguably less significant, complication is that the penalty functions can cause the boundary of the constraint to soften. As candidate solutions continuously move from the inside to the outside of each constraint's feasible region, the value of the penalty continuously increases, which may result in an optimum near a constraint boundary being considered slightly less preferable, or a candidate solution slightly outside the constraint boundary being considered slightly more preferable. The degree of softness of each constraint's penalty can be adjusted by tuning the values of $k_1, \ldots, k_n$. This softness is acceptable for the virtual cinematography applications considered in this work, as none of the constraints considered are highly sensitive to candidate solutions precisely respecting the feasible region boundary.

4.2 Optimization Algorithms

There are a myriad of techniques for computing an optimum for a continuous optimization problem. If the objective function is of one of several simple forms, such as linear or quadratic, then it may be possible to analytically solve the optimization problem analytically. However, many if not most of the optimization problems presented here have no such guarantees. Most are nonlinear, others non-convex, and others still inexpressible as closed form functions (see Section 7.2.2.2).

In cases such as these, a numerical optimization algorithm is best suited, although

Algorithm 1: Gradient Descent

```
Given:
    $\mathbf{x^0}$: Initial function input
    $\gamma$: Learning rate constant
    $N$: Maximum number of iterations
1 while $||\nabla f(\mathbf{x}^i)|| \geq \epsilon$ and $i < N$ do
2     $\mathbf{x}^{i+1} \leftarrow \mathbf{x}^i - \gamma \nabla f(\mathbf{x}^i)$
3     $i \leftarrow i + 1$
```

choosing a specific algorithm can be challenging.

If the given objective function is relatively smooth and convex, a simple gradient descent (GD) algorithm can be used to find a relatively precise approximation of the optimum, as is outlined in Algorithm 1. However, if the objective function is non-convex, this method is susceptible to becoming stuck in local minima. Additionally, selecting an appropriate value for γ can be tricky. Too small a value will waste valuable computation time, while too large a value can cause severe oscillations if the magnitude of the gradient changes quickly. An additional consideration is that gradient descent requires the gradient of the function to be calculated for each iteration, which may be computationally costly for complicated objectives functions.

One option to mitigate gradient descent's sensitivity to γ and reliance on repeated gradient computations is to instead use a backtracking line search (BLS) algorithm, as outlined in Algorithm 2. By searching along a single gradient direction for a better candidate at each iteration, the algorithm can take larger steps without needing more gradient computations. However, the backtracking line search algorithm may only provide a small computational speed boost over gradient descent, as the algorithm may request an excessive number of function evaluations without careful tuning of its hyperparameters. Additionally, as it is fundamentally gradient based, backtracking line search is prone to becoming stuck at local minima.

Stochastic algorithms such as simulated annealing (SA) provide a potential solution to the problem of local minima. There are a number of different formulations of the general simulated annealing algorithm, but we used the version outlined in Algorithm 3 in our

Algorithm 2: Backtracking Line Search

```
Given:
    $\mathbf{x}^0$: Initial function input
    $\gamma_0$: Initial Learning Rate
    $\alpha$: Expected improvement of the function relative to the gradient
    $\beta$: Rate of change to learning rate on improvement failure
    $N$: Maximum number of gradient updating iterations
    $N_s$: Maximum numer of backtracking iterations
1 $i \leftarrow 0$
2 while $||\nabla f(\mathbf{x}^i)|| > \epsilon$ and $i < N$ do
3     $j \leftarrow 0$
4     $\gamma \leftarrow \gamma_0$
5     while $f(\mathbf{x}^i + \gamma \nabla f(\mathbf{x}^i)) > f(\mathbf{x}^i) + \alpha ||\nabla f(\mathbf{x}^i)||^2$ and $j < N_s$ do
6         $\gamma \leftarrow \beta\gamma$
7         $j \leftarrow j + 1$
8     $\mathbf{x}^{i+1} \leftarrow \mathbf{x}^i - \gamma \nabla f(\mathbf{x}^i)$
9     $i \leftarrow i + 1$
```

experiments. While simulated annealing is far less sensitive to local minima, the solutions it produces are far less precise than those found by gradient based methods if the global minima is found.

Another stochastic algorithm popular in virtual cinematography research is particle swarm optimization (PSO), with the specific version used in our experiments outlined in Algorithm 4. While more complex and computationally expensive than simulated annealing, PSO seems to compute solutions in the neighborhood of global optima with higher probability than simulated annealing.

While I have also experimented with simplex (Nelder-Mead) methods, I have so far been disappointed by their performance. It might be to do with varying sensitivities over different variable types (e.g., position vs. rotation), or with the inherently modular nature of rotation variables, or some other hyperparameter tuning.

Generally speaking, the approach taken in the experiments has been to begin with a stochastic algorithm to roughly identify the neighborhood of the global optimum, then improve the precision of the result with a gradient based algorithm initialized. Additionally,

Algorithm 3: Simulated Annealing

```
Given:
    $\mathbf{x}_0$: Initial function input
    $T_0$: Initial temperature
    $C$: Cooling rate where $0 < C < 1$
    $R(a, b)$: Uniformly distributed random function where $a \leq R(a, b) \leq b$
1 $i \leftarrow 0$
2 while $T_i > 1$ do
3     $\mathbf{x}^{i+1} \leftarrow \mathbf{x}^i + R(-T_{i+1}, T_{i+1})$
4     $T_{i+1} \leftarrow T_i(1 - C)$
5     if $\exp\left(\frac{f(\mathbf{x}^i) - f(\mathbf{x}^{i+1})}{T_i}\right) \geq R(0, 1)$ then
6         $\mathbf{x}^{i+1} \leftarrow \mathbf{x}^i$
7     $i \leftarrow i + 1$
```

when attempting to compute the next keyframe in a series, the search can be initialized to begin in the neighborhood of the previous keyframe's optimum. Both of these approaches are forms of warm start optimization.

A key feature of all of these algorithms is that each can be tuned to run in any amount of computational time, with a tradeoff between higher precision results and faster completion. This flexibility allows these same algorithms to be used for both online and offline applications.

4.3 Temporal Smoothing

Simple temporal summation over an instantaneous objective function as $\sum f(\mathbf{x}(t))$ completely fails to model the inertia and momentum viewers expect from real world cameras. Generally speaking, any camera path that is not smooth tends to be perceived as unnervingly artificial and mechanical. This issue can be addressed by extending our given model with a smoothing penalty based on an active contour model (Kass et al., 1988). In this augmented model, the system attempts to minimize the energy functional

$$E(\mathbf{x}(t)) = E_{\text{int}}(\mathbf{x}(t)) + E_{\text{ext}}(\mathbf{x}(t)) \tag{4.1}$$

Algorithm 4: Particle Swarm Optimization

```
Given:
    $\mathbf{x}_0$: Initial function input
    $n_p$: Number of particles
    $n_g$: Number of generations
    $c_a$: Individuality coefficient
    $c_b$: Sociality coefficient
    $\mathbf{R}(c)$: $\mathbb{R} \to \mathbb{R}^d$ Uniformly distributed random vector function where
    $-c \leq \mathbf{R}(c)_i \leq c \forall i \in [1, d]$
1  foreach $i \in 1, ..., n_p$ do
2      $\mathbf{v}_i \leftarrow \mathbf{R}(something)$
3      $\mathbf{x}_i \leftarrow \mathbf{x}_0 + \mathbf{R}(something)$
4      $\mathbf{p}_i \leftarrow \mathbf{x}_0$
5      if $f(\mathbf{x}_i) < f(\mathbf{g})$ then
6          $\mathbf{g} \leftarrow \mathbf{x}_i$
7  while $t < n_g$ do
8      foreach $i \in 1, ..., n_p$ do
9          $\mathbf{v}_i \leftarrow \omega \mathbf{v}_i + \mathbf{R}(c_a) \odot (\mathbf{p}_i - \mathbf{x}_i) + \mathbf{R}(\mathbf{c}_b) \odot (\mathbf{g} - \mathbf{x}_i)$
10         $\mathbf{x}_i \leftarrow \mathbf{x}_i + \mathbf{v}_i$
11         if $f(\mathbf{x}_i) < f(\mathbf{p}_i)$ then
12             $\mathbf{p}_i \leftarrow \mathbf{x}_i$
13             if $f(\mathbf{x}_i) < f(\mathbf{g})$ then
14                 $\mathbf{g} \leftarrow \mathbf{x}_i$
15     $t \leftarrow t + 1$
16 return $\mathbf{g}$
```

comprising an internal energy

$$E_{\text{int}}(\mathbf{x}(t)) = \frac{1}{2}\left(\alpha|\dot{\mathbf{x}}(t)|^2 + \beta|\ddot{\mathbf{x}}(t)|^2\right) \tag{4.2}$$

and an external energy

$$E_{\text{ext}}(\mathbf{x}(t)) = f(\mathbf{x}(t)), \tag{4.3}$$

where α and β are given constants, and the overstruck dots denote differentiation with respect to time t.

Including the smoothness of the path in the evaluation model comes with a beneficial side effect. If no satisfactory solution exists for a portion of the optimal solution for

$E_{\text{ext}}(\mathbf{x}(t))$, the internal energy of the active contour model allows for the camera path to smoothly interpolate between the known better solutions at neighboring times. Additionally, the ability of this method to consider the changing value of the objective function while smoothing allows for the preservation of higher priority camera behaviors while compromising lower priority behaviors, a property not shared by methods that smooth output paths as a blind post-processing step after optimization.

Determining suitable values for α and β generally requires experimental parameter tuning.[78] If either is set too high, the resultant camera path may violate the desired objectives, while too low a setting may fail to alleviate the undesirable artificiality. Furthermore, there is no requirement that the same constants are desirable for all variables. For example, a user who wishes their camera to rotate rather than move may set a higher α or β for camera position variables than for rotation variables.

Unfortunately, simple attempts at numerically solving the active contour model can be hideously inefficient, especially for large values of α and β. If derivatives of the equation above are naively computed, it may take $O(n^2)$ iterations of gradient descent, where n is the number of keyframes, for an impulse to traverse from one end of the chain of keyframes to the other.

Instead, the standard method of computation is to reformulate the problem using the Euler-Lagrange equation, where any minimum of the original equation 4.1 is also a solution to

$$\alpha\ddot{\mathbf{x}}(t) - \beta\ddddot{\mathbf{x}}(t) - \nabla E_{\text{ext}}(\mathbf{x}(t)) = 0. \tag{4.4}$$

Solving this numerically generally requires that the considered path is discretized to a sequence of n keyframes. Notationally, we can assume that $\mathbf{x}(i) = \mathbf{X}_{i,*}$, meaning that $X_{i,j}$ corresponds to the jth component of the ith keyframe. Put another way, X is a n by m matrix, where each row corresponds to all input variables of a particular keyframe, while each column of X corresponds to all keyframes of a particular input variable. For notational brevity, $\mathbf{X}_i$ is used to denote $\mathbf{X}_{i,*}$ in the below.

[78]Experimentally, it was found that $\alpha = 0.01$, $\beta = 0.2$, and $c(t) = 1$ produced decent simulation results.

Additionally, the keyframe sequence is not assumed to be uniformly distributed across time, with the time interval between pairs of consecutive keyframes varying throughout the sequence. This can be captured by supposing that the sequence of keyframes is associated with a corresponding sequence of time states $\mathbf{t}$, with keyframe i corresponding to time t_i and $t_{i-1} < t_i < t_{i+1}$.

In this format, the temporal derivatives of $\mathbf{x}(t)$ can be expressed in finite form as

$$\dot{\mathbf{x}}(t) = \dot{\mathbf{X}}_i = \frac{\mathbf{X}_{i+1} - \mathbf{X}_i}{t_{i+1} - t_i}, \tag{4.5}$$

$$\ddot{\mathbf{x}}(t) = \ddot{\mathbf{X}}_i = \frac{\mathbf{X}_{i+1} - \mathbf{X}_i}{t_{i+1} - t_i} - \frac{\mathbf{X}_i - \mathbf{X}_{i-1}}{t_i - t_{i-1}}, \tag{4.6}$$

$$\ddddot{\mathbf{x}}(t) = \ddddot{\mathbf{X}}_i = \frac{\ddot{\mathbf{X}}_{i+1} - \ddot{\mathbf{X}}_i}{t_{i+1} - t_i} - \frac{\ddot{\mathbf{X}}_i - \ddot{\mathbf{X}}_{i-1}}{t_i - t_{i-1}}. \tag{4.7}$$

From these finite forms, equation 4.4 can be rewritten as

$$MX - \nabla \sum_{t=1}^{n} E_{\text{ext}}(X_{*,t}) = 0, \text{ where } M = A(\alpha L - \beta L^2)B. \tag{4.8}$$

Here, L is an $(n+2) \times (n+2)$ matrix corresponding to a padded Laplacian operator, directly following equation 4.6, as

$$L = \begin{pmatrix} 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ \frac{1}{t_2-t_1} & -\frac{2}{t_2-t_1} & \frac{1}{t_2-t_1} & 0 & \cdots & 0 & 0 & 0 \\ 0 & \frac{1}{t_2-t_1} & -\left(\frac{1}{t_3-t_2}+\frac{1}{t_2-t_1}\right) & \frac{1}{t_3-t_2} & \cdots & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{t_3-t_2} & -\left(\frac{1}{t_4-t_3}+\frac{1}{t_3-t_2}\right) & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & -\left(\frac{1}{t_n-t_{n-1}}+\frac{1}{t_{n-1}-t_{n-2}}\right) & \frac{1}{t_n-t_{n-1}} & 0 \\ 0 & 0 & 0 & 0 & \cdots & \frac{1}{t_n-t_{n-1}} & -\frac{2}{t_n-t_{n-1}} & \frac{1}{t_n-t_{n-1}} \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \end{pmatrix}, \tag{4.9}$$

while A is an $n \times (n+2)$ matrix, which can be interpreted as eliminating the padding,

$$A = \begin{pmatrix} 1 & 1 & 0 & 0 & \ldots & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & \ldots & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & \ldots & 0 & 1 & 1 \end{pmatrix}. \tag{4.10}$$

However, there are multiple options as to the $(n+2) \times n$ padding matrix B, which differ based on the intended partial differential equation boundary condition. If the user intends the desired path to begin and end with the camera stationary before accelerating, with $\mathbf{X}_0 = \mathbf{X}_1$ and $\mathbf{X}_n = \mathbf{X}_{n+1}$ corresponding to a classic Neumann differential equation boundary condition, a value of

$$B = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & 0 & \dots & 0 & 1 \\ 0 & 0 & 0 & \dots & 0 & 1 \end{pmatrix} \tag{4.11}$$

should be chosen. However, if the user intends the desired path to begin and end with the camera already moving at a constant velocity, with $\dot{\mathbf{X}}_0 = \dot{\mathbf{X}}_1$ and $\dot{\mathbf{X}}_n - 1 = \dot{\mathbf{X}}_n$, this padding matrix can be encoded correspondingly as

$$B = \begin{pmatrix} 2 & -1 & 0 & \dots & 0 & 0 \\ 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & 0 & \dots & 0 & 1 \\ 0 & 0 & 0 & \dots & -1 & 2 \end{pmatrix}. \tag{4.12}$$

An iterative numerical solver generally following the structure of gradient descent can be derived from this form of the problem. Given an estimate X^i, a lower energy estimate

$$X^{i+1} = (I - \gamma M)^{-1} \left(X^i - \gamma \nabla \sum_{t=1}^{n} E_{\text{ext}}(\mathbf{X}_t^i) \right) \tag{4.13}$$

for some search step size γ. This implicit solving formulation allows for much larger step sizes than any explicit solver would, even with high values of α and β. To reach a stable equilibrium representing a solution to the underlying problem, this iteration should continue until $|X^{i+1} - X^i| < \epsilon$. Note that the matrix inversion required by this formulation can be performed as an LU decomposition, which is relatively computationally expensive. As a result, changing the value of γ between optimization iterations, as would be desirable for a backtracking line search algorithm, is usually not practical.

As an alternative, an explicit solver could compute increasingly accurate estimates as

$$X^{i+1} = X^i - \gamma \left(M X^i + \nabla \sum_{t=1}^{n} E_{\text{ext}}(\mathbf{X}_{*,t}^i) \right). \tag{4.14}$$

However, this requires more iterations and a lower choice of γ to reach a stable equilibrium, generally carrying a higher computational cost.

However, some care must be taken for search variables representing modulo domains, such as Euler angle rotation representations. In these cases, consecutive keyframes must be normalized to ensure the direction and magnitude of differences is sensible, such that $X^0_{i,k} \leftarrow X^0_{i,k-1} + (X^0_{i,k-1} - X^0_{i,k}) \bmod n_i$ for component specific mod bases $\mathbf{n}$, before the active contour model gradient descent can begin. Specifically, this modulo behavior needs to correspond to rounding division, where the remainder $r = a \bmod n$ such that $a = nq + r$ for $q = \text{round}\left(\frac{a}{n}\right)$.

4.4 Use Cases

While there are many different ways of utilizing the cinematographic tools outlined above, the system supports two primary categories of use cases, which are labeled as *scripted* and *unscripted* scenarios.

4.4.1 Scripted Scenarios

In a scripted scenario, the system attempts to find a satisfactory camera path for a preplanned scene before any part of it is to be displayed to the viewer. As this is an offline operation, the system can take as much time as is necessary to find a satisfactory solution. Furthermore, complete knowledge of all past, current, and future states of the scene are available at every instant of time. This is analogous to the traditional preproduction workflow of a live action or animated film.

Scripted scenarios can be solved in a straightforward manner by the optimization strategy described in Section 4.2. The general strategy I have followed is to first optimize each frame individually, using SA or PSO followed by GD or BLS to refine the results. If smoothing is desired, this is followed by smoothing with the implicit gradient descent formulation described in Section 4.3.

4.4.2 Unscripted Scenarios

In an unscripted scenario, the goal of the system is to find a camera pose for the current instant in time given complete knowledge of the current and past states of the scene, but without any direct knowledge of the future. This is an online operation during the playback of the scene; therefore, the system must be able to find a satisfactory solution in real time given the viewer's desired frame rate. This is analogous to the shooting of an unscripted documentary or news material, or to video game camera control. Of course, unscripted scenarios can also be simulated by playing scripted motions back in real-time.

Without smoothing by an active contour model, the first frame camera pose is set either by the user or by running SA or PSO, then GD or BLS is run every frame starting from the values of the previous frame, a technique commonly referred to as warm start.

Unscripted scenarios with smoothing are significantly trickier, since in order for the active contour model to work effectively, the system must know something of the unknown future. As our current system is lacking contextual or behavioral data, as employed by (Halper et al., 2001) to form predictions more precisely, the prediction scheme currently supported is a Taylor series approximation where, for future keyframes $t_1, \ldots, t_n$,

$$\mathbf{x}(t_j) \approx \sum_{k=0}^{n} \frac{1}{k!}(t_j - t_0)^k \frac{d^k\mathbf{x}(t_0)}{dt^k}, \tag{4.15}$$

where $d^k\mathbf{x}(t)/dt^k$ is the k^{th}-order derivative of $\mathbf{x}(t)$. To alleviate instability stemming from discrete time steps, the system calculates the necessary derivatives as local averages

$$\frac{d^k\mathbf{x}(t)}{dt^k} \approx \frac{1}{M\Delta_t} \sum_{i=0}^{M} \frac{d^{k-1}\mathbf{x}(t-\Delta_t i)}{dt^{k-1}} - \frac{d^{k-1}\mathbf{x}(t-\Delta_t(i+1))}{dt^{k-1}}, \tag{4.16}$$

where $\Delta_t = t_i - t_{i+1}$. Of course, this is a tradeoff as too large a value of M can create an undesirable sense of variable inertia. Values of $M = 10$ and $\Delta_t = 0.1\,\text{sec}$ were used.

Once these predictions are made, the same types of optimization techniques can be used to find a suitable camera path through the predicted period, and the first future keyframe $\mathbf{x}(t_1)$ can be output as the next camera state.

CHAPTER 5

Computational Tools Systems Engineering

Thus far, this document has focused solely on the abstract mathematical concepts involved with declarative virtual cinematography in offline and online domains. Of course, experimenting with these concepts in practice requires a concrete software system with particular requirements.

At a high level, an idealized imagining of system operation would only require the user to specify an objective function and leave it to the system's tools to compute the optimum. However, many of the optimization techniques mentioned in the previous sections require computational and analytical capabilities that go beyond simple function evaluation: gradient descent requires that the derivatives of the objective function be calculated, while hierarchical penalties requires knowledge of the range of functions, to give two examples.

Requiring the user to manually supply instructions for each of these computation types would dramatically increase the amount of user effort required to use the system. Worse still, a user hoping to iteratively experiment with different objective functions in such a manually instructed system would have to correctly change the instructions for each type of computation, where any typo or incongruity in one set of instructions can cause incredibly difficult to debug behavior.

Instead, a more user friendly and efficient model would include automated functionality to derive these instructions for these different computation types given only the original objective function, with these derivations ideally occurring rapidly at runtime to allow for the rapid prototyping of objective functions.

Supporting both offline and online applications brings another set of system engineering challenges. These derived operations must not only be automatic, but also efficient enough to be run in between frame updates. Additionally, the system must be able to be integrated into a real time graphics system, interacting with whatever geometric query and control infrastructure is foundational to the underlying system.

These requirements can be more tidily summarized as follows:

1. Support for the evaluation of arbitrary objective functions, including black box expressions incorporating knowledge of the broader state of the scene (e.g., raycasts, collision tests).[79]

2. Support for static computation of the range of an objective function, sometimes called value range analysis, in support of the hierarchical constraints described in Section 4.1.1.

3. Support for the automatic evaluation of function gradients, often called automatic differentiation, for gradient based optimization methods.[80]

4. Runtime scriptability for rapid prototyping of different objective functions.

5. Ability to be easily integrated into the loop of a real time graphics system (e.g., Unreal Engine 4 game engine), running based on scenes that are either live (online) or based on recorded data (offline).

When I first began this project in 2015, I spent a considerable amount of time and energy searching for an existing software system that met all of these requirements.

While a fully featured computer algebra system (CAS), such as in MATLAB or SageMath, could easily satisfy requirements 2, 4, and 3, satisfying requirements 1 and 5 proved remarkably elusive. My attempts to integrate black box expressions such as raycasts in the form of networking between the CAS and main graphics systems seemed to introduce significant latency issues. Despite my best efforts, I could only get the CAS

[79]This is necessary for visibility and collision objectives. See Section 7.2.2.2.

[80]Such as gradient descent. See Section 4.2.

to return consistent results of a controllable quality **or** achieve real time interactivity within a graphics loop, but not both in the same attempt.[81]

While any embedded implementation of an interpreted programming, such as Python, JavaScript, or Lua, language would certainly be able to support requirement 4 as well as 1, and might be able to support requirement 5, the other requirements are a bit trickier. While there are a variety of publicly available automatic differentiation packages that might partially satisfy requirement 3, all that I examined were relatively limited, requiring objective functions to be specified using a narrower set of operations than required to implement my desired functions. Worse still, I could find no existing libraries that supported automatic value range analysis for either Python, JavaScript, or Lua, making requirement 2 impossible using existing tools.[82]

A third alternative considered was to use scripts written in a compiled language, such as Java or C++, for which existing automatic differentiation and value range analysis are more commonly available. However, supporting runtime scriptability, requirement 4, is difficult with these compiled languages, requiring some type of just in time compilation and/or dynamic loading, and made even more difficult by requiring support for black box functions that incorporate information from the running real time graphics system. As a result, I estimated that the cost of integration using compiled languages would be far greater than incrementally developing a bespoke system, and with a far greater lead time before even the simplest of results could be seen.[83]

[81]Whether this is an inherent incompatibility or simply the result of my own inability is unclear. MATLAB is certainly capable of real time performance on its own, and even has an official package featuring plugins for network-like interfacing with game engines. However, the application domains of these real time packages and interfaces are incredibly narrow, whereas the type of optimization based computation I was attempting is far more generic. Perhaps this would be worth revisiting if a more restricted form of computation was desired in the future.

[82]The reader might ask why I didn't instead focus my efforts on developing these missing tools, given the lack of any other inherent issue with this approach. At the time, I estimated that the cost of development of these tools for an existing language such as Python would be greater than creating a bespoke system. As the project progressed and the true magnitude of my chosen approach became clear, my confidence in this early estimate has shrunk considerably.

[83]Just as before, my estimate has probably been proven false over the ensuing development period.

As a result of the apparent fruitlessness of this search, I made the fateful[84] decision to build my own system, largely from scratch.[85]

At a high level, the core of the resulting system operates like a small, imperative, statically-typed programming language. The basic operation cycle can be summarized as follows: a program corresponding to instructions to evaluate a desired objective function is specified by the user, the program is then analyzed and transformed to support alternate computation types as needed to support the type of optimization desired, then the final resulting program is interpreted within a real time graphics environment.

The first version of this system, written to implement the concepts described in (Litteneker and Terzopoulos, 2017), featured $\sim$ 6.5K lines of hastily written, inflexible C++ code. In the years since, this codebase has adapted and expanded into $\sim$ 20K lines of modular C++ and Lua code, integrated into an Unreal Engine 4 (Epic Games, 2022) project. The codebase is itself integrated with several further open source libraries, including the following.

- Runtime scriptability is achieved through an integration with Lua (Ierusalimschy, 2006), which is bound to the C++ object-oriented API using sol2 (ThePhD, 2018).
- Eigen (Guennebaud et al., 2010) is integrated to support hardware accelerated operations on vectors, matrices, and quaternions.
- While many optimization algorithms are my own implementations in C++, several come from a header-only integration with the optimlib library (O'Hara, 2020).

While the specifics of this codebase are inexorably mixed with the idiosyncrasies of C++, in particular making heavy use[86] of template metaprogramming, as well as those

[84]As well as (arguably) incredibly foolhardy.

[85]Internally, the system library is referred to as *AUL*. I wish I could say it stands for something novel, like "Advanced Utility Language" or "Alan's Unusual Library," but it actually just stands for my first, middle, and last initials. Instead of exercising in what feels like a strange act of vanity by filling this chapter with references to my own name, I have instead chosen to simply refer to it as the "system" throughout this document.

[86]Some might call it abuse.

of the other integrated libraries, many of the algorithms and data structures employed to support the requirements outlined above are easily generalizable to a far simpler language model, which is described in detail below. While the full system is a fiercer and less tidy beast, the micro-language discussed here is more than sufficient to facilitate concise discussion of the automatic differentiation, value range analysis, expression simplification, and black box function integration tools at the heart of the full system.

5.1 Micro Language Specification

For the purposes of illustration, we examine a small, imperative programming language designed to primarily operate over scalars (n).[87] However, in order to fully support the transformations and analyses to be discussed later, we also need stacks of numbers (ℓ) as well as stacks of statements (r). Formally, these data types can be denoted as follows:

n ::= *<real number literal>*

ℓ ::= () | (n, ℓ)

r ::= [] | [s, r]

v ::= n | ℓ

Programs in this language can be expressed following the grammar below. In addition to categories of variables (x), statements (s), and expressions (e), the language also supports black-box functions accepting a single numeric argument (f). While a formal definition for these is omitted, f functions can be assumed to correspond to the address of some external, side-effect free function.

x ::= *<variable identifier>*

f ::= *<black box function accepting a single numeric argument>*

s ::= skip | s ; s | x = e | e

    | while e < e do s | break

[87]While this has no bearing on the rest of this discussion, it is interesting to note that this micro-language is Turing complete.

```
    |  if $e$ < $e$ then $s$ else $s$
    |  pushv($e$, $x$) | popv($x$)
    |  pushr($s$, $x$) | execr($s$, $x$)
$e$ ::= $n$ | $x$ | $s$ : $e$
    |  $e$ + $e$ | $e$ - $e$ | $e$ * $e$ | $e$ / $e$ | $e^n$
    |  peekv($e$) | $f$($e$)
$t$ ::= $v$ | $s$ | $e$
```

While many of the above terms, such as assignment, sequence statements, and arithmetic expressions, likely appear familiar to even a novice coder, there are a few unusual features deserving of brief comment. Both `if` and `while` statements are supported, but both are limited to inequality conditions, purely for simplicity.[88] `break` statements are supported, allowing for The : operator, called the sequence expression, allows for any expression to be preceded by a statement with side effects.

The `pushv`, `popv`, and `peekv` terms facilitate the manipulation and accessing of stacks of numbers to a variable in memory. Specifically, `pushv` and `popv` allow for a single number to be pushed or popped from the top of a stack, while `peekv` allows for the topmost value in the stack to be accessed. While an empty stack of numbers can be assigned to a variable, the only way of manipulating and accessing values stored in that stack are through `pushv`, `popv`, and `peekv`, each of which is capable of modifying only one value at a time.[89]

Perhaps the most unusual are the `pushr` and `execr` terms, which operate over stacks of statements, and are invaluable in supporting certain forms of automatic differentiation in 5.4.2. To give a brief overview of the intended usage of these terms, `execr(`s`,`x`)` first initializes x to correspond to an empty stack of statements in the program memory, executes some statement s, then executes whatever statements have been pushed to x by

[88] General boolean conditions and expressions are supported in the full system, but detailing all of the additional terms and semantics would cause this already lengthy section to grow longer and more complex.

[89] This is not a strict requirement of the tools and techniques used here

`pushr` statements during the execution of s.

There are a few additional properties of the semantics of this language that are useful to note.

Firstly, expressions can have side effects. For example, the expression `x=x+5: x*x` not only depends upon the incoming value of x but also modifies the outgoing value of x.

Secondly, the order of evaluation for all executable programs is well defined, including over arithmetic expressions. The program `x=2: (x=x*x: x)*(x=x*3: x)` should have a resulting value of 48, and should have a side effect that the value of `x` be 12 after execution.

Thirdly, stacks of statements, values of r, can only be manipulated by `execr` and `pushr` statements. The types of side effects these allow are also limited, restricted to the initialization of an empty stack in memory or the pushing of a single statement onto an existing stack in memory. Stacks of statements in memory cannot be modified in any other way, including by swapping or duplicating through assignment to another variable.

While the full system supports a simple strict-type checker[90], programs in the language will be assumed to be well-formed. Formal rules for such a system have been omitted here in the interest of brevity, but can be easily derived in a similar manner as Pierce and Benjamin (2002).[91]

Finally, a program in the language described above can be modeled as a directed acyclic graph (DAG). The semantics of this language can further be read as the foundation for a graph walking interpreter, with the execution of each term depending only on the incoming program state and this term's immediate children, which is how the current interpreter for this language is currently implemented.

[90]This type checker naturally arises from C++ polymorphism over term types, as expression types are valid.

[91]One of the reasons this would become so long is their technique requires small-step semantics, which would roughly double the number of semantic judgements required.

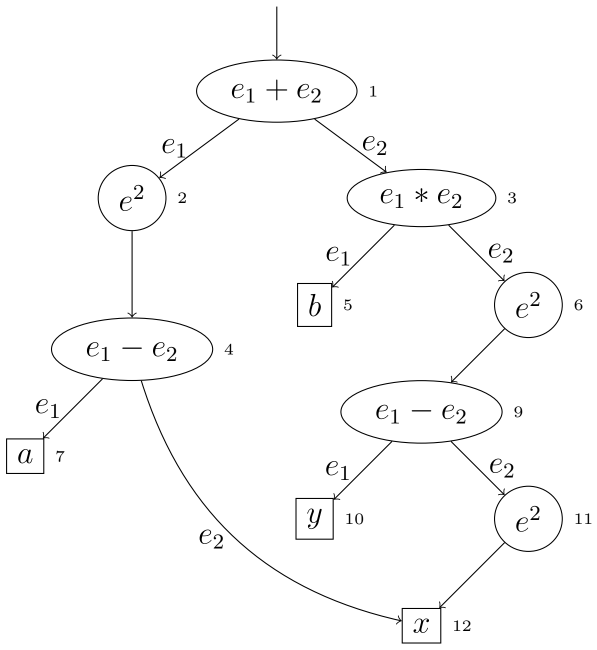


Figure 5.1: Graph of a program that represents the Rosenbrock function.

As an example, consider the Rosenbrock function,

$$f(x, y) = (a - x)^2 + b(y - x^2)^2 \tag{5.1}$$

which has a global minimum at $x = a$ and $y = a^2$ and is commonly used for testing optimization systems (Rosenbrock, 1960). A program in the micro-language for evaluating the Rosenbrock function can be represented in the graph pictured in Figure 5.1.

5.2 Semantics

Borrowing the imperative big step style from (Kirchner and Sinot, 2007) (which is, in turn, adapted from (Pierce and Benjamin, 2002) among others), we can define a set of formal rules for this simple language's operational semantics below.

These semantics are expressed as judgments over operations of the form $\langle t_1, \sigma_1 \rangle \Downarrow \langle t_2, \sigma_2 \rangle$, where σ corresponds to the state of the program memory at each big-step of execution. The value of variable x in σ is denoted as $\sigma[x]$, while a modification of the value of x in σ to value v is denoted as $\sigma\{x \mapsto v\}$.[92]

Specifically, step operations for statements follow a restricted form of $\langle s, \sigma_1 \rangle \Downarrow_s \langle s \in \{\texttt{skip}, \texttt{break}\}, \sigma_2 \rangle$, with `skip` and `break` serving as signals for loop control flow. As expressions can step to any value, their operations have a slightly different form of $\langle e, \sigma_1 \rangle \Downarrow_e \langle v, \sigma_2 \rangle$. Step operations over statements and expressions are disambiguated by their subscripts.

The operational judgments for "familiar" statements can be written as follows:

$$\frac{}{\langle \texttt{skip}, \sigma \rangle \Downarrow_s \langle \texttt{skip}, \sigma \rangle} \qquad \frac{}{\langle \texttt{break}, \sigma \rangle \Downarrow_s \langle \texttt{break}, \sigma \rangle}$$

$$\frac{\langle e, \sigma \rangle \Downarrow_e \langle v, \sigma' \rangle}{\langle x \texttt{:=} e, \sigma \rangle \Downarrow_s \langle \texttt{skip}, \sigma`\{x \mapsto v\} \rangle} \qquad \frac{\langle e, \sigma \rangle \Downarrow_e \langle v, \sigma' \rangle}{\langle e, \sigma \rangle \Downarrow_s \langle \texttt{skip}, \sigma` \rangle}$$

$$\frac{\langle s_1, \sigma \rangle \Downarrow_s \langle \texttt{t}, \sigma' \rangle \qquad t \neq \texttt{skip}}{\langle s_1 \texttt{;} s_2, \sigma \rangle \Downarrow_s \langle \texttt{t}, \sigma' \rangle} \qquad \frac{\langle s_1, \sigma \rangle \Downarrow_s \langle \texttt{skip}, \sigma' \rangle \qquad \langle s_2, \sigma' \rangle \Downarrow_s \langle t, \sigma'' \rangle}{\langle s_1 \texttt{;} s_2, \sigma \rangle \Downarrow_s \langle t, \sigma'' \rangle}$$

$$\frac{\begin{array}{c}\langle e_1, \sigma \rangle \Downarrow_e \langle n_1, \sigma' \rangle \qquad \langle e_2, \sigma' \rangle \Downarrow_e \langle n_2, \sigma'' \rangle \\ n_1 < n_2 \qquad \langle s, \sigma'' \rangle \Downarrow_s \langle \texttt{break}, \sigma''' \rangle\end{array}}{\langle \texttt{while}\ e_1\ \texttt{<}\ e_2\ \texttt{do}\ s, \sigma \rangle \Downarrow_s \langle \texttt{skip}, \sigma''' \rangle}$$

$$\frac{\begin{array}{c}\langle e_1, \sigma \rangle \Downarrow_e \langle n_1, \sigma' \rangle \qquad \langle e_2, \sigma' \rangle \Downarrow_e \langle n_2, \sigma'' \rangle \qquad n_1 < n_2 \\ \langle s, \sigma'' \rangle \Downarrow_s \langle \texttt{skip}, \sigma''' \rangle \qquad \langle \texttt{while}\ e_1\ \texttt{<}\ e_2\ \texttt{do}\ s, \sigma'' \rangle \Downarrow_s \langle \texttt{skip}, \sigma''' \rangle\end{array}}{\langle \texttt{while}\ e_1\ \texttt{<}\ e_2\ \texttt{do}\ s, \sigma \rangle \Downarrow_s \langle \texttt{skip}, \sigma''' \rangle}$$

$$\frac{\langle e_1, \sigma \rangle \Downarrow_e \langle n_1, \sigma' \rangle \qquad \langle e_2, \sigma' \rangle \Downarrow_e \langle n_2, \sigma'' \rangle \qquad n_1 \geq n_2}{\langle \texttt{while}\ e_1\ \texttt{<}\ e_2\ \texttt{do}\ s, \sigma \rangle \Downarrow_s \langle \texttt{skip}, \sigma'' \rangle}$$

[92]For any readers unfamiliar with the style of judgments used here, the premise below the line is guaranteed to be true if all premises above the line are true. For example, the judgment

$$\frac{premise_1\ \ premise_2}{conclusion}$$

could be read that if both premises are true, the conclusion must also be true.

$$\frac{\langle e_1, \sigma\rangle \Downarrow_e \langle n_1, \sigma'\rangle \qquad \langle e_2, \sigma'\rangle \Downarrow_e \langle n_2, \sigma''\rangle \\ n_1 < n_2 \qquad \langle s_1, \sigma''\rangle \Downarrow_s \langle t, \sigma'''\rangle}{\langle \texttt{if } e_1 \texttt{ < } e_2 \texttt{ then } s_1 \texttt{ else } s_2, \sigma\rangle \Downarrow_s \langle t, \sigma'''\rangle}$$

$$\frac{\langle e_1, \sigma\rangle \Downarrow_e \langle n_1, \sigma'\rangle \qquad \langle e_2, \sigma'\rangle \Downarrow_e \langle n_2, \sigma''\rangle \\ n_1 \geq n_2 \qquad \langle s_2, \sigma''\rangle \Downarrow_s \langle t, \sigma'''\rangle}{\langle \texttt{if } e_1 \texttt{ < } e_2 \texttt{ then } s_1 \texttt{ else } s_2, \sigma\rangle \Downarrow_s \langle t, \sigma'''\rangle}$$

$$\frac{\langle e, \sigma\rangle \Downarrow_e \langle n, \sigma'\rangle \qquad \ell = \sigma'[x]}{\langle \texttt{pushv(}e\texttt{, } x\texttt{)}, \sigma\rangle \Downarrow_s \langle \texttt{skip}, \sigma'\{x \mapsto (n, \ell)\}\rangle}$$

$$\frac{(n, \ell) = \sigma[x]}{\langle \texttt{popv(}x\texttt{)}, \sigma\rangle \Downarrow_s \langle \texttt{skip}, \sigma'\{x \mapsto \ell\}\rangle}$$

Expressions can be written similarly, as below. Note that the similar binary arithmetic terms of +, -, *, and / are unified under a single category of op, with their associated mathematical operation written as [[op]], in the interest of brevity.

$$\frac{}{\langle n, \sigma\rangle \Downarrow_e \langle n, \sigma\rangle} \qquad \frac{}{\langle \ell, \sigma\rangle \Downarrow_e \langle \ell, \sigma\rangle} \qquad \frac{v = \sigma[x]}{\langle x, \sigma\rangle \Downarrow_e \langle v, \sigma\rangle}$$

$$\frac{(n, \ell) = \sigma[x]}{\langle \texttt{peekv(}x\texttt{)}, \sigma\rangle \Downarrow_e \langle n, \sigma\rangle} \qquad \frac{\langle e, \sigma\rangle \Downarrow_e \langle n_1, \sigma`\rangle \qquad n_2 = f(n_1)}{\langle f\texttt{(}e\texttt{)}, \sigma\rangle \Downarrow_e \langle n_2, \sigma`\rangle}$$

$$\frac{\langle e_1, \sigma\rangle \Downarrow_e \langle n_1, \sigma'\rangle \qquad \langle e_2, \sigma'\rangle \Downarrow_e \langle n_2, \sigma''\rangle \qquad n_3 = n_1[[\text{op}]]n_2}{\langle e_1 \texttt{ op } e_2, \sigma\rangle \Downarrow_e \langle n_3, \sigma''\rangle}$$

$$\frac{\langle e, \sigma\rangle \Downarrow_e \langle n_2, \sigma'\rangle \qquad n_3 = n_2^{n_1}}{\langle e^{n_1}, \sigma\rangle \Downarrow_e \langle n_3, \sigma'\rangle} \qquad \frac{\langle s, \sigma\rangle \Downarrow_e \langle \texttt{skip}, \sigma'\rangle \qquad \langle e, \sigma'\rangle \Downarrow_e \langle v, \sigma''\rangle}{\langle s\!:\!e, \sigma\rangle \Downarrow_e \langle v, \sigma''\rangle}$$

It should be noted that, as a result of the notational style used here, break statements with no while statement ancestors separating them from a sequence expression cause undefined behavior. In other words, the program `x=5;while x<10 do x=(break:x)+1` would **not** be executable using these specific semantics. Fortunately, this is trivial to check before attempting execution by constructing a control flow graph as described in Section 5.3.1, and checking whether the `break` statement has any successors.

The remaining terms deal with accessing and manipulating stacks of statements. To give a brief overview of the intended usage of these terms, $\texttt{execr(}s\texttt{,}x\texttt{)}$ first initializes x to correspond to an empty stack of statements in the program memory, executes some statement s, then executes whatever statements have been pushed to x by `pushr` statements during the execution of s. Operational semantics corresponding to these terms can be formalized as follows:

$$\frac{r = \sigma[x]}{\langle \texttt{pushr(}s\texttt{, }x\texttt{)}, \sigma\rangle \Downarrow_s \langle \texttt{skip}, \sigma\{x \mapsto \texttt{[}s, r\texttt{]}\}\rangle}$$

$$\frac{\langle s, \sigma\{x \mapsto \texttt{[]}\}\rangle \Downarrow_s \langle \texttt{skip}, \sigma'\rangle \qquad r = \sigma'[x] \qquad \langle r, \sigma'\rangle \Downarrow_s \langle \texttt{skip}, \sigma''\rangle}{\langle \texttt{execr(}s\texttt{, }x\texttt{)}, \sigma\rangle \Downarrow_s \langle \texttt{skip}, \sigma''\rangle}$$

$$\frac{\langle s, \sigma\rangle \Downarrow_s \langle \texttt{skip}, \sigma'\rangle \qquad \langle r, \sigma'\rangle \Downarrow_s \langle \texttt{skip}, \sigma''\rangle}{\langle \texttt{[}s\texttt{, }r\texttt{]}, \sigma\rangle \Downarrow_s \langle \texttt{skip}, \sigma''\rangle} \qquad \frac{}{\langle \texttt{[]}, \sigma\rangle \Downarrow_s \langle \texttt{skip}, \sigma\rangle}$$

5.3 Static Analysis

Supporting value range analysis, one of the original requirements for the system, as well as a few other performance related program queries, requires a set of powerful **static analysis** tools. At a high level, these tools can be understood as answering one or more of the following questions **without** evaluating any of the original program:

- Control Flow Analysis: In what order can terms in this language be evaluated?
- Value Range Analysis: What values can a variable or expression evaluate to under a set of assumptions about incoming variable values?

- Program Optimization: Are there any terms in the existing program that can be replaced with more efficient but semantically identical terms?

Techniques for answering these questions are far from unprecedented. Various forms of control flow analysis and compiler optimization have been used in a variety of applications since at least the 1970s, while value range analysis has been a semi-regular components of some compilers since the 1990s (Aho et al., 2006).

Despite this precedence, developing efficient implementations of these tools for a language with features as unusual as thus micro-language is a nontrivial task.

5.3.1 Control Flow Analysis

Analyzing the order of evaluation of terms in this micro-language takes the form of constructing a control flow graph (CFG), where a directed edge between two nodes indicates that the execution of the tail node may immediately precede the execution of the head node. While traditional CFGs are constructed with nodes corresponding to atomic program nodes, supporting the relatively complex variety in operational semantics for the different types of terms in this micro-language requires something slightly more complex.

Instead, each term corresponds to a pair of nodes in the CFG, with the `start` and `end` nodes respectively corresponding to the beginning and completion of evaluation of a given term. In this formulation, local control flow edges are formed by edges from parents' start nodes to childrens' start nodes, between childrens' start and end nodes, and from childrens' end nodes to parents' end nodes.

Formally, the control flow graph, $G = (V, E)$, has the same vertices regardless of term types or structure, with

$$V = \{p | p \in \{t :: start, t :: end\}, t \in \text{program terms}\}.$$

Denoting that a control flow graph contains an edge from a tail node a to a head node b

Term	Constraints
`skip` $\mid n \mid \ell \mid t \mid x \mid$ `popv(`x`)`	E $\subseteq$ start $\rightarrow$ end
s_1 `;` s_2	E $\subseteq$ start $\rightarrow$ s_1::start E $\subseteq$ s_1::end $\rightarrow$ s_2::start E $\subseteq$ s_2::end $\rightarrow$ end
`while` e_1 `<` e_2 do s	E $\subseteq$ start $\rightarrow$ e_1::start E $\subseteq$ e_1::end $\rightarrow$ e_2::start E $\subseteq$ e_2::end $\rightarrow$ s::start E $\subseteq$ s::end $\rightarrow$ e_1::start E $\subseteq$ e_2::end $\rightarrow$ end
`if` e_1 `<` e_2 then s_1 else s_2	E $\subseteq$ start $\rightarrow$ e_1::start E $\subseteq$ e_1::end $\rightarrow$ e_2::start E $\subseteq$ e_2::end $\rightarrow$ s_1::start E $\subseteq$ e_2::end $\rightarrow$ s_2::start E $\subseteq$ s_1::end $\rightarrow$ end E $\subseteq$ s_2::end $\rightarrow$ end
s `:` e	E $\subseteq$ start $\rightarrow$ s::start E $\subseteq$ s::end $\rightarrow$ e::start E $\subseteq$ e::end $\rightarrow$ end
e_1 `op` e_2	E $\subseteq$ start $\rightarrow$ e_1::start E $\subseteq$ e_1::end $\rightarrow$ e_2::start E $\subseteq$ e_2::end $\rightarrow$ end
$e^n \mid x$ `:=` $e \mid$ `pushv(`e`,` x`)` $\mid$ `peekv(`e`)`	E $\subseteq$ start $\rightarrow$ e::start E $\subseteq$ e::end $\rightarrow$ end
`break`	E $\subseteq$ start $\rightarrow$ { frontier of `while` ancestors accessible using only `if` and sequence terms }

Table 5.1: Standard Control Flow Graph Construction Rules

can be written as $E \subseteq a \rightarrow b$.

Following this notation, the local CFG edges for the “familiar” term types can be shown in table 5.1.

However, this leaves `execr` and `pushr`, which instead executes statements that have been individually pushed onto a statement stack in memory. As the order of execution of these stack pushed statements itself depends on the order of their being pushed, constructing CFG edges over these terms is instead an exercise in querying or transforming the structure of the existing control flow graph.

One concise way of expressing this operation is to define a helper function $\phi(G, n, x, \texttt{bool})$, which returns the frontier of `pushr` terms pushing to x that are reachable from node n by traversing edges in G forward if the last argument is `false` or backward if `true`. Using this function, the edges for `pushr` and `execr` terms can be expressed following the rules in table 5.2.

Term	Constraints
`pushr(`s`, `x`)`	$E \subseteq$ start $\rightarrow$ end $\forall t \in \phi(G, \text{start}, x, false).E \subseteq s :: \text{end} \rightarrow t\text{::start}$
`execr(`s`, `x`)`	E $\subseteq$ start $\rightarrow$ s::start $\forall t \in \phi(s\text{::end}, x, false).E \subseteq s\text{::end} \rightarrow t\text{::start}$ $\forall t \in \phi(s\text{::start}, x, true).E \subseteq t\text{::end} \rightarrow \text{end}$

Table 5.2: CFG construction rules for non-standard terms.

An important feature of these constraints is that they are recursive: what edges should appear in the graph depends on what edges are already in the graph. As a result, constructing a graph that satisfies all of these constraints is somewhat nontrivial.

The solution I employed was a fixed point algorithm, where the constraints were iteratively applied until an iteration found no additional edges to be necessary. While this theoretically might require $O(|V|^2)$ iterations to reach the fixed point, all programs used in experimentation for this paper required no more than three iterations: the first iteration added the constant edges for the "familiar" terms, the second added the edges for `execr`/`pushr` terms, and the third iteration found no additional edges to add.

However, each iteration of this algorithm may be as expensive as $O(|V|^2)$, as computing ϕ may be $O(|V|)$. This can be accelerated considerably by instead querying a contracted form of G, where an edge $n_1 \rightarrow n_2$ can be contracted iff neither n_1 nor n_2 corresponds to a `execr`/`pushr` term. Running this contraction once at the beginning of each iteration requires only $O(|E|)$, while ϕ function queries over this contracted graph operate in constant time. Therefore, the cumulative computation and memory complexity of this control flow graph construction is generally linear in the number of terms in the program.

One final note, each CFG node p is annotated with what variables are referenced,

$refs(p)$, and mutated, $muts(p)$, at that node. This is useful for clarifying and disambiguating between the effects between different phases of execution of each term.

Term	References	Mutations
`x`	$refs(\text{ end }) = \{x\}$	
`x = e`		$muts(\text{ end }) = \{x\}$
`pushv(x, e)`	$refs(\text{ start }) = \{x\}$	$muts(\text{ end }) = \{x\}$
`popv(x)`	$refs(\text{ start }) = \{x\}$	$muts(\text{ end }) = \{x\}$

Table 5.3: Rules for Side Effect Static Analysis

Together with the CFG, these $refs$ and $muts$ annotations provide sufficient information for a form of data dependency analysis. A CFG node p_a may directly depend upon the side effects of CFG node p_b if every path from p_b to p_a spanning $\{p_1, \ldots, p_n\}$ satisfies $refs(p_a) \cap muts(p_b) \cap \bigcup_{i=1}^{n} muts(p_i) \neq \emptyset$. For notational simplicity, this dependency query is denoted as $dep(p_a, p_b)$.

Additionally, these $refs$ and $muts$ annotations can be trivially extended to terms, with $refs(t) = refs(t :: start) \cup refs(t :: end) \cup \bigcup_{t' \in children(t)} refs(t')$ and $muts(t) = muts(t :: start) \cup muts(t :: end) \cup \bigcup_{t' \in children(t)} muts(t')$. While this loses some of the fine granularity of the CFG annotations, having the ability to generalize these annotations to terms allows for term level analysis that can be useful in later analyses.

5.3.2 Value Range Analysis

The form of value range analysis used here computes the possible ranges of expressions and variables as real numeric **intervals**.[93] Possible interval values include both those of the form $[a, b]$ for any $a, b \in \mathbb{R}$ such that $a \leq b$[94], as well as $\emptyset$, which is necessary for the tracking of intermediate results.

Arithmetic operations and exponentiation corresponding to the types of expressions

[93]Why intervals? For many programs, there are an infinite, or at least astronomical, number of possible values, far too many to tractably enumerate let alone store. Intervals provide an encapsulation that is efficient to both store and manipulate using operations derived from interval arithmetic. Additionally, the stated use case of this analysis, the hierarchical constraints described in Section 4.1.1, require nothing more than intervals.

[94]This can be read that any element of $\{x \in \mathbb{R} | a \leq x \leq b\}$ is possible.

included in this language have corresponding operations over intervals, as outlined below. Each of these interval arithmetic operations produces a single interval output which contains all possible results of applying the corresponding mathematical operation to any combination of singular real number values within the ranges of the interval operands.[95].

$$\begin{aligned}
-[a,b] =&[-b,-a]\\
[a_1,b_1]+[a_2,b_2] =&[a_1+a_2, b_1+b_2]\\
[a_1,b_1]-[a_2,b_2] =&[a_1-b_2, b_1-a_1]\\
[a_1,b_1]\cdot[a_2,b_2] =&[\min(a_1a_2,a_1b_2,b_1a_2,b_1b_2),\max(a_1a_2,a_1b_2,b_1a_2,b_1b_2)]\\
\frac{[a_1,b_1]}{[a_2,b_2]} =&\begin{cases}[a_1,b_1]\cdot[\frac{1}{b_2},\frac{1}{a_2}] & \text{if } 0\notin[a_2,b_2]\\ [a_1,b_1]\cdot[\frac{1}{b_2},\infty] & \text{if } a_2=0, b_2>0\\ [a_1,b_1]\cdot[-\infty,\frac{1}{a_2}] & \text{if } a_2<0, b_2=0\\ [-\infty,\infty] & \text{if } a_2<0<b_2\\ \emptyset & \text{if } a_2=b_2=0\end{cases}\\
[a,b]^n =&\begin{cases}[0,\max(a^n,b^n)] & \text{if } n\%2=0, a\le 0\ge b\\ [\min(a^n,b^n),\max(a^n,b^n)] & \text{otherwise}\end{cases}
\end{aligned}$$

If at least one of the operands of any of the interval arithmetic operations described above is $\emptyset$, the result is $\emptyset$.

$$-\emptyset = [a,b]\ \texttt{op}\ \emptyset = \emptyset\ \texttt{op}\ [a,b] = \emptyset^n = \emptyset$$

Additionally, the hull of two intervals, sometimes referred to as the least upper bound,

[95]While reasoning about intervals began in antiquity, this style of interval arithmetic was invented in the 20th century. While some of its principals were first introduced by Young (1931) and others in the 1930s, modern interval analysis largely stems from the work of mathematician Ramon E. Moore, and his highly influential Moore (1966).

can be simply defined as

$$\begin{aligned}
[a_1, b_1] \cup [a_2, b_2] &= [\min(a_1, a_2), \max(b_1, b_2)], \\
[a, b] \cup \emptyset = \emptyset \cup [a, b] &= [a, b], \\
\emptyset \cup \emptyset &= \emptyset.
\end{aligned}$$

Using these properties and operations, an inequality constraint between two interval variables can be defined. Some interval x *contains* interval y, written as $x \subseteq y$, if and only if $x \cup y = x$.

Note that arithmetic operations over intervals follow the same commutativity, associativity, and distributivity properties as their real valued counterparts. One subtle but significant technical point is that $\pm\infty \cdot 0 = 0$ in interval operations, as $\pm\infty$ values refer to the limit of a range of finite values rather than any truly infinite value.

Analyzing a program using interval based value range analysis can be expressed as a data flow analysis problem. In this form, the values of interval variables corresponding to the ranges of expressions and variables are allowed to flow through constraints corresponding to local program behavior. By iteratively expanding the interval variables until all constraints are satisfied, the resulting ranges should be no larger than necessary to capture all possible values. This approach is also a fixpoint algorithm, of a similar form to that used in control flow graph construction in the previous section.[96]

It is important to note that, while all possible values must be contained within the computed ranges for the analysis to be valid, the computed ranges may contain values that are not strictly possible. Some of this redundancy is the unavoidable result of a lack of cognizance within the interval arithmetic operations of any dependency between operands.

[96]This dataflow-fixpoint based approach for analyzing programs was invented by Gary Kildall as part of his 1972 PhD dissertation, and is sometimes referred to as the Kildall method. The domain of his work was primarily over discrete lattices of values, rather than intervals or control flow graphs, but the general approach is fundamentally similar. Kildall was a computer scientist and entrepreneur who made substantial technical contributions to compilers, operating systems, media hardware, and more in the early days of personal computers before his sudden death at the age of 52 (Markoff, 1994).

For example, if x is assumed to have a range of $[-1, 1]$ then applying the arithmetic rules above to the expression $x * x$ would produce a range of $[-1, 1]$, ignoring the fact that both factors are guaranteed to have the same sign. While a slightly more sophisticated analysis might produce the true range of this simple example, only marginally more complex expressions can suffer from dependency problems running from computationally expensive to undecidable. Worse still, some interval arithmetic functions, such as division, can produce infinite ranges depending on operand value.[97]

Two different forms of value range analysis are supported, separated by their cognizance of the order of evaluation of terms, a property known as **flow sensitivity**.

5.3.2.1 Flow Insensitive

In a flow insensitive analysis, the order of evaluation of terms is ignored. While this can significantly reduce the precision of the resulting intervals for programs heavily utilizing side effects, the analysis itself is simpler to implement, faster to compute, and far less memory intensive.

A flow insensitive analysis problem operates over two sets of intervals. The first corresponds to the potential ranges of each variable in the program, denoted $vars[x]$ for each variable x in the program. The second corresponds to the potential result range of each expression, and is denoted $res[e]$ for each expression e in the program. Note that these intervals capture both numbers, n, as well as lists of numbers, ℓ, as interval ranges. Stacks of statements, r, are not captured in this analysis, but their behavior should already be fully captured in the control flow graph. The problem of flow insensitive value range analysis can then be expressed as interval inequality constraints over values in these two sets of variables.

The constraints over $vars[x]$ are very simple. As this analysis does not incorporate any

[97]Division in particular is sometimes written as resulting in a pair of intervals if the divisor crosses zero. This is more precise, but any expression tree containing n levels of nested divisions would then produce 2^n output intervals, which can quickly become computationally intractable. While the single interval result used here is less precise, its constant memory and computational footprint maintains tractability across the sort of large and sophisticated expressions for which this software system is intended.

order of evaluation information into the analysis, the only terms that need be considered are those that can introduce another value to a scalar or numeric stack variable. Namely, these terms are assignment, $x{=}e$, and pushing a number onto a stack, $\texttt{pushv}(x, e)$. Each appearance of these terms can correspond to an inclusion of the following constraint, with x and e corresponding to the relevant children of each type of term.

$$vars[x] \subseteq res[e]$$

The constraints for $res[e]$ can likewise be simply expressed using the rules for interval arithmetic outlined in the previous section.

$$\begin{aligned}
res[\texttt{[]}] &\subseteq \emptyset \\
res[n] &\subseteq [n, n] \\
res[x] &\subseteq vars[x] \\
res[s : e] &\subseteq res[e] \\
res[peekv(x)] &\subseteq vars[x] \\
res[e_1 \ \texttt{op} \ e_2] &\subseteq res[e_1] \ \texttt{op} \ res[e_2] \\
res[e^n] &\subseteq res[e]^n \\
res[f(e)] &\subseteq [-\infty, \infty]
\end{aligned}$$

With $vars[x]$ and $res[e]$ initialized to $\emptyset$ for all appropriate x, e, the goal is to find the minimum ranges for each variable that satisfy all constraints corresponding to terms in the program. A theoretically sound way to compute this is to increase the size of the left hand side interval variable of each constraint until it contains or is equal to the interval value of the right hand side, and repeat this process until all constraints are satisfied, a state known as the fixed-point.

However, a simple implementation of this may probe intractable depending on the type of program. As the expansion of these ranges is iterative and the maximum size of each range is infinite, it is possible that an infinite number of finite expansions may be necessary to reach the fixed-point. Reducing this infinite number of iterations can be accomplished by **widening** ranges to infinity if they appear to be incrementally expanding across long periods of iterations. For example, consider the program `x=0; while x < f(x) do x = x + 1`. Each iterative application of the constraints may expand the range of x by 1 but the fixed-point of the algorithm has the range of x as $[0, \infty]$. If the range of x at iterations i and $i + \Delta$, for some sufficiently large Δ, appears to be increasing, we can proactively widen the range of x to $[0, \infty]$ to force a fixed-point for x. While this carries some risk of unnecessarily expanding ranges further than necessary if Δ is chosen unluckily, widening is the only reliable method of preventing infinite (or astronomical if implemented with floating point numbers) runtime before reaching the fixed-point.[98]

A valuable property to note of is that each of these constraints is local, depending only on the ranges corresponding to child terms and local variable references/mutations. As a result, we do not need to consider all constraints at every iteration, as only those constraints that depend on recently modified ranges will produce any new range expansions. This can be easily supported by keeping a queue of constraints to apply, and adding absent constraints to this queue if they depend on ranges modified by the constraint currently being applied. This approach is sometimes called a **work-list**, and can provide a considerable runtime speedup, as much as 100x in my experience (Aho et al., 2006).

However, even with widening and the work-list approach, the runtime required to reach a fixed-point is nontrivial. A program with N terms can clearly have no more than N variables, the *var* and *res* range variables for which are related by no more than N constraints as each term in the program may correspond to no more than one constraint. Therefore, it must be the case that the first possible range variable requiring widening must be discoverable after $O(N)$ steps, as to suppose otherwise would require that some interval variable be expanded more than once without requiring widening. If, in the worst

[98]The general technique of widening was first introduced for static analysis over discrete lattice domains in (Cousot and Cousot, 1977).

Algorithm 5: Value Range Analysis

```
Given:
    $\mathbf{X}$: Set of interval variables $x_1, \dots, x_m$ representing domain
    $\mathbf{C}$: Inequality constraints $c_1, \dots, c_n$ where $c_i$ is $x_{c_i} \subseteq e_{c_i}$
    $q$: Number of iterations between widening considerations
1  $j \longleftarrow 0$
2  $\mathbf{Y} \longleftarrow \mathbf{X}$
3  $\mathbf{W} \longleftarrow \mathbf{C}$ as a stack
4  while $workList$ is not empty do
5      $c \longleftarrow$ popped front element of $\mathbf{W}$
6      $v \longleftarrow e_c$
7      if $x_c \not\subseteq v$ then
8          $x_c \longleftarrow x_c \cup v$
9          foreach $c_j \in \mathbf{C}$ that references $x_c$ do
10             if $c_j \notin \mathbf{W}$ then
11                 push $c_j$ on to back of $\mathbf{W}$
12     $j \longleftarrow j + 1$
13     if $j\%q = 0$ then
14         foreach $x_i \in X$ do
15             if $Y_i \neq \emptyset$ and $Y_i \not\subseteq x_i$ then
16                 $x_i \longleftarrow [-\infty, \infty]$
17                 foreach $c_j \in \mathbf{C}$ that references $x_i$ do
18                     if $c_j \notin \mathbf{W}$ then
19                         push $c_j$ on to back of $\mathbf{W}$
20         $\mathbf{Y} \longleftarrow \mathbf{X}$
```

case, widening must be used for all variables in the program, with the number of variables having an upper bound of N and widening considered every N iterations, then the upper bound on the runtime of this analysis must be $O(N^2)$.

Nevertheless, some programs require a much lower runtime for analysis. Programs containing few terms with side effects, such as the Rosenbrock function described in equation 5.1, or those with few loops in the constraint variables, such as programs written in static single assignment form, can have flow insensitive value range analysis complete in as little as $O(N)$. In practice, most programs I have experimented with tend to be closer to $O(N)$ than $O(N^2)$, but the runtime clearly varies based on specific program structure and form.

5.3.2.2 Flow Sensitive Analysis

Contrastingly, a flow sensitive analysis fully considers the order of evaluation of terms, specifically over the control flow graph in my formulation. As the converse to the previous subsection, flow sensitive analysis is much more precise than flow insensitive, especially in the context of programs with more complex dependency graphs, but comes at the cost of much higher computational cost.

As each control flow graph node can correspond to a change in the environment state, each node p has a corresponding set of input and output ranges for all variables x, denoted as $in[p, x]$ and $out[p, x]$ respectively. Just as with flow insensitive analysis, the interval corresponding to the result of each expression e is denoted as $res[e]$.

With $in[p, x]$, $out[p, x]$, and $res[e]$ initialized to $\emptyset$ for all appropriate p, x, e, the problem of value range analysis becomes a problem of finding the minimum ranges that satisfy constraints of the following forms. Note that many of the res constraints are the same as those already listed for flow insensitive analysis, with only those for variable reference and `peekv` requiring redefinition. Each term corresponds to exactly one in and out constraint, while each expression additionally corresponds to one res constraint.

$$\frac{in[p,x] \subseteq \bigcup_{q \in prev(p)} out[q,x]}{}$$

$$out[x_1 = e :: end, x_2] \subseteq \begin{cases} res[e] & \text{if} x_1 = x_2 \\ in[x_1 = e :: start, x_2] & \text{if} x_1 \neq x_2 \end{cases}$$

$$out[\texttt{pushv}(x_1, e) :: end, x_2] \subseteq \begin{cases} in[\texttt{pushv}(x_1, e) :: start, x_2] \cup res[e] & \text{if} x_1 = x_2 \\ in[\texttt{pushv}(x_1, e) :: start, x_2] & \text{if} x_1 \neq x_2 \end{cases}$$

$$\frac{out[p,x] \subseteq in[p,x] \text{ for all other } p, x}{}$$

$$\begin{aligned} res[x] &\subseteq in[x :: start, x] \\ res[\texttt{peekv}(x)] &\subseteq in[\texttt{peekv}(x) :: start, x] \\ \vdots \quad &\vdots \quad \vdots \end{aligned}$$

One subtle but important property of the *in* and *out* constraints outlined above is that they relate sets of interval variables together. As a program with N terms has an upper bound of N variables, the full constraint problem will have $O(N^2)$ variables connected by $O(N^2)$ constraints between individual variables.

Even with the widening and work-list accelerations described in the previous section, I was experimentally finding that solving this required an intractable amount of memory. This can be significantly improved by noting that the majority of the constraints, especially between *in* and *out* ranges, are likely to simply copy ranges from another node. This is especially true with programs that have linear control flow graphs with few side effects, such as the Rosenbrock function discussed earlier. In cases such as these, considerable savings can be made by using a linked flyweight design pattern (Gamma et al., 1995), where each range can either be unique or linked to the value of another range. This both reduces the number of stored values, shrinking the memory footprint, and lowers the number of values that must be checked with each constraint application, speeding up the execution of the analysis.

Following the same logic described in the previous section, flow sensitive value range analysis has a runtime complexity with an upper bound of $O(N^4)$ and a lower bound of

$O(N^2)$, with significant variations between those two stemming from different program structures and forms. Even in the most charitable case, evaluating flow sensitive value range analysis is at least as expensive as its flow insensitive counterpart, with a possibility of a much higher cost depending on program structure.

5.3.3 Program Simplification and Caching

One of the greatest practical uses of the analysis forms already outlined lies in finding opportunities to remove redundancies in a given program. Program simplification, alternatively called program optimization, is a common technique among compilers, and often represents one of the most expensive computations performed as part of the compilation process. While the micro-language specified here is interpreted rather than compiled, providing automatic tools for program simplification is vital to providing even remotely efficient automatic differentiation, as well as more broadly accelerating the performance of human input programs.

The program simplification analysis described here is, like many of the previously described forms of analysis, implemented as a fixed-point algorithm, with local simplification rules repeatedly applied when their conditions allow until no more simplifications can be found. This analysis is subdivided into several phases, separated by the types of analysis results required to evaluate each phase:

A Using only the **program graph**, there are several easily detectable program simplifications, including the following:

  A1 Removal of `skip` statements from sequence and `pushr` statements:

  - $\texttt{skip};s \to s$
  - $s;\texttt{skip} \to s$
  - $\texttt{skip}:e \to e$
  - $\texttt{pushr}(\texttt{skip}, x) \to \texttt{skip}$

A2 Removal of redundant assignments in which no reference to the assigned variable appears in the program: $(x = e) \rightarrow$ `skip` if $\forall t.x \notin refs(t)$

A3 Removal of redundant `pushv`/`popv` without a corresponding `peekv` or other reference to the stack variable:
`pushv`/`popv`$(x, e) \rightarrow$ `skip` if $\forall t.t \notin \{\texttt{pushv}, \texttt{popv}\}, x \notin refs(t)$

A4 Removal of side effect free statements: $s \rightarrow$ `skip` if $muts(s) = \emptyset$

An important property of these rules to note is that, while each rule is local, the application of a rule can make other rules subsequently applicable. For example, the program `x=2;  2` can be simplified first to `skip;  2` by rule A2, and then to `2` by rule A1.

Additionally, as each of these rules can be computed in constant time, assuming $\{t|x \in refs(t)\}$ is precomputed, a single iteration of this phase of simplification analysis can be computed in $O(N)$, where N is the number of terms in the program. As the application of each rule can reduce the size of the program by 1, no more than N iterations can be performed before reaching a fixed-point. As a result, running this phase of analysis to a fixed-point has a theoretical runtime complexity of $O(N^2)$. However, as not every term type has an associated simplification rule to consider, the actual computation time required is highly dependent on the program structure. Empirically, this phase is the fastest of the phases described here.

B Utilizing **flow insensitive value range analysis** results, a variety of expression related simplifications are available, including the following:

B1 Constant expression simplification: $e \rightarrow n$ if $res[e] = [n, n]$ and $muts(e) = \emptyset$

B2 Addition identity simplification: $(e_1+e_2) \rightarrow e_1$ if $res[e_2] = [0, 0]$ and $muts(e_2) = \emptyset$, $(e_1 + e_2) \rightarrow e_2$ if $res[e_1] = [0, 0]$ and $muts(e_1) = \emptyset$

B3 Subtraction identity simplification: $(e_1 - e_2) \rightarrow e_1$ if $res[e_2] = [0, 0]$ and $muts(e_1) = \emptyset$

B4 Multiplication identity simplification: $(e_1 * e_2) \to e_1$ if $res[e_2] = [1, 1]$ and $muts(e_2) = \emptyset$, $(e_1 * e_2) \to e_2$ if $res[e_1] = [1, 1]$ and $muts(e_1) = \emptyset$

B5 Division identity simplification: $(e_1/e_2) \to e_1$ if $res[e_2] = [1, 1]$ and $muts(e_2) = \emptyset$

While a single iteration of these rules on a program with N terms is certainly executable in $O(N)$, their dependence on first evaluating flow insensitive value range analysis adds an additional computational overhead of somewhere between $O(N)$ and $O(N^2)$. Therefore, a single iteration of this phase of simplification is theoretically at least as slow as phase A, and empirically tends to be a fair bit slower.

Rules in this phase can make rules in this phase and the previous phase subsequently applicable. For example, the program $x = 2 : x$ can be simplified to $x = 2 : 2$ by rule B1, then to 2 by the steps outlined in the previous section. Can any of the rules in this phase make rules in this same phase subsequently applicable? I suspect not, but I'm not 100% sure.

C By analyzing the **control flow graph** for the entire program, a more precise form of redundant assignment removal rule can be established.

The rule specified in A2 misses redundant assignments where subsequent assignments shadow the result to any variable references that follow. For example, the simple program $x = 1; x = 2 : x$ clearly has a redundant assignment, $x = 1$, that would not be simplifiable under the rules previously outlined. We can intuitively imagine a new rule that tests whether a given assignment has any references it could affect, which can be formally defined as $(x = e) \to \texttt{skip}$ if $\nexists p \in G.x \in refs(p), p$ is reachable from $(x = e) :: end$ by traversing only CFG nodes q where $x \notin muts(q)$.

While this rule is far more powerful than A2, evaluating this condition requires a full depth first search of the CFG, which may have a runtime complexity of $O(N)$. As a result, applying this phase to a program with many assignments may be $O(N^2)$

for a single iteration.[99] Unlike previous phases, this theoretical runtime analysis seems to better match the observed performance characteristics, and is certainly slower than any of the previous phases.

In addition, rules in this phase *can* produce simplifications that can make rules in this phase and preceding phases applicable. For example, the program $x = 1; x = 2 : x$ is simplifiable to $\texttt{skip}; x = 2 : x$ by rule C, then to $x = 2 : x$ by rule A1, to $x = 2 : 2$ by rule B1, to $\texttt{skip} : 2$ by rule A2, and finally to 2 by C.

D Finally, utilizing **flow sensitive value range analysis** can provide the most powerful simplification rules of all, including the following:

D1 Redundant sequential assignments simplification: $(x = e) \to e$ if $in[x = e, x] = out[x = e, x]$ and $out[x = e, x] = [a, a]$. While this rule is valuable, a useful property to note is that it is only applicable in programs where subsequent assignments are to the same singular value (e.g., $x = 1; x = 1$) but not to programs where subsequent assignments are relative to the value of some unknown value (e.g., $x = f(1); x = f(1)$).

D2 Removal of constant value stack terms:

- $\texttt{pushv}(e, x) \to e$ if $out[\texttt{pushv}(e, x), x] = [a, a]$
- $\texttt{popv}(x) \to \texttt{skip}$ if $out[\texttt{popv}(e, x), x] = [a, a]$
- $\texttt{peekv}(x) \to \texttt{a}$ if $out[\texttt{peekv}(x), x] = [a, a]$

Note that all of these must trigger for the program to remain semantically unchanged, as any dangling `pushv` or `popv` terms may cause the stack variable to grow or shrink more than it did in the original program, potentially leading to undefined behavior.

[99] I actually prototyped definition-use (DU) and use-definition (UD) chain analyses to try to speed this up. While it can provide a speed up in programs with relatively few variables relative to the number of assignments, most of the programs I have experimented with are on the other side of that ratio. Unfortunately, this means that UD/DU analyses provide little to no runtime savings, yet can cost a great deal of memory. Perhaps some method of piecemeal UD/DU analysis would provide a reliable speedup, but I have not explored this. Perhaps if the program could be first subdivided into relatively independent regions in a more efficient way, then each region could be analyzed independently.

D3 All of the rules for flow insensitive value range analysis (B) are equally applicable in this phase. The additional precision flow sensitivity provides allows for programs such as $x = 1; x = x + x : x$ to be simplified.

Continuing the trend, this phase is more computationally expensive than any of the other phases, as the prerequisite flow sensitive value range analysis requires somewhere between $O(N^2)$ and $O(N^4)$ depending on program structure.

Just as before, rules in this phase can make rules in this phase and the previous phase subsequently applicable. For example, the program $x = 1; x = x + x : x$ can be simplified to $x = 1; x = 2 : x$ by utilizing a flow sensitive version of B1, then to 2 by the steps outlined in the previous section.

Structuring all of these phases into a single efficient simplification algorithm is somewhat unintuitive.

As already noted, the goal of this fixed-point algorithm is to repeatedly run the phases of local simplification rules until no further simplifications can be found. However, each phase has a different computational cost, with higher phases carrying a higher computational cost than any lower phase. Additionally, each simplification phase is capable of making lower level simplification phases applicable, and many simplification phases also allow subsequent applications of the same phase to produce further simplifications.

Of separate note is the order of evaluation of simplifications. Many of the simplification rules described require there to be strict alignment between the preprocessing computation and the evaluation of the condition. For example, the program $x = 3 : y * (x - 2)$ has several applicable simplifications in phase D, of which we can consider two: $(x - 2) \to (1)$ by the phase D equivalent of rule B1, and $y * (x - 2) \to y$ by the phase D equivalent of rule B3. If $(x - 2) \to (1)$ was evaluated *and* applied first, any attempt at evaluating the latter rule requires $res[1]$ which was not part of the original value range analysis and would not be defined. As a result, all applicable simplifications for a particular phase must be computed and cached before any program transformations are performed.

However, caching these simplifications requires some care. As these transformations are local to each term, and many simplifications reduce to something derived from a child term, simplifications must be applied to parents before children for all to apply. For example, $0 + (0 + x)$ has two simplifications available from rule B2 or its equivalent in phase D: $0 + (0 + x) \to (0 + x)$ and $(0 + x) \to (x)$. If $0 + (0 + x) \to (0 + x)$ was performed *before* $(0 + x) \to (x)$, the latter would be effectively undone by the former: $0+(0+x) \to 0+(x) \to (0+x)$, rather than the more efficient $0+(0+x) \to 0+(0+x) \to (x)$, and another iteration of the phase would be required to reach the fixed-point

A simplification algorithm following all of these desired properties can be formulated as algorithm 6.

Algorithm 6: Program Simplification

```
function SimplifyPhase($phase$):
    Run whatever precomputation is necessary for this phase
    $S \longleftarrow emptystack$
    foreach $t \in$ program in topological order do
        push applicable simplifications for $t$ in $phase$ to back of $S$
    foreach $s \in S$ from front to back do
        apply $s$
    return True if $S$ is non-empty, False if $S$ is empty
changed = True
while $changed$ do
    while $changed$ do
        while $changed$ do
            while $changed$ do
                $changed$ = SimplifyPhase(A)
            $changed$ = SimplifyPhase(B)
        $changed$ = SimplifyPhase(C)
    $changed$ = SimplifyPhase(D)
```

Finally, the entire program can be further accelerated by performing common subexpression elimination style caching. As there is no guarantee that the program would conform to static single assignment form, techniques such as value numbering are not available. Instead, the problem is solved by considering pairs of syntactically (i.e., graphically) equivalent terms, and testing whether every path through the control flow graph between the considered terms lack mutations to any variables referenced by the considered terms. The computation associated with this analysis is expensive, in many cases more so than another of the previous phases in simplification, and that cost is greater for larger programs. However, it does not require iteration to reach a fixed point: one iteration is enough. As a result, common subexpression elimination is computed only after all other simplification is completed, and the program has reached some minimal size.

5.4 Automatic Differentiation

Broadly speaking, automatic differentiation, abbreviated hereafter to **autodiff**, is the problem of automatically computing derivatives, over specific input variables for a given expression program. While autodiff may be relatively simple in theory, finding an *efficient* and *precise* computational approach represents a thoroughly nontrivial task in practice, as many implementers have learned.[100]

Consider an abstract mathematical expression $f : \mathbb{R} \to \mathbb{R}$ of the form

$$f(x) = f_1(f_2(f_3(\ldots f_{n-1}(x) \ldots))).$$

While there are a variety of ways of implementing an evaluation procedure for this function practically, all reasonable implementations must evaluate f_{i+1}, …, f_{n-1} before evaluating f_i.

Derivatives for this function could be estimated numerically, for example by finite differences as $\frac{df}{dx} = \frac{f(x)-f(x+\Delta)}{\Delta}$ for some chosen Δ. In some rare cases, such as with black box functions in the micro-language, a numerical differentiation method may be the only

[100]Autodiff has a long but murky history. Alonzo Church was probably the first to reference the abstract notion of automatic differentiation of computer programs, but he stopped well short of anything like a concrete autodiff formulation (Church, 1941). There are some accounts, such as (Graham, 2009), describing a lecture John McCarthy gave in 1959 where he wrote code on a black-board for automatic differentiation of LISP programs, but all of these accounts are from long after the event and McCarthy never seems to have published this work. The individual with the earliest independently confirmable claim to inventing/discovering autodiff of which I am aware is Robert Edwin Wengert, whose short (only 2 pages) paper introduced simple but practical formulations of both forward and reverse accumulation for Fortran programs (Wengert, 1964). However, most modern historical accountings of machine learning, where autodiff is most studied at present, usually cite Linnainmaa (1970), Rumelhart et al. (1986), or Le Cun and Fogelman-Soulié (1987) as originating modern back-propagation, the specialized form of reverse accumulation autodiff currently ubiquitous in GPU parallelized machine learning. I have so far been unable to find a citation chain from these back-propagation papers back to the work of Wengert or those that followed him, suggesting that autodiff might have been independently discovered/invented at least twice.

viable method of differentiation. However, this numerical approach is both inefficient, as the function must be evaluated multiple times for each derivative computation, and imprecise, as poor choices of Δ can cause the estimation to ignore small function features or introduce floating point precision errors.

Alternatively, we could consider symbolic differentiation, the task of analytically finding a separate expression representing the derivative of the given function relative to each of its input variables, mirroring the pen-and-paper technique typically taught in university or secondary school calculus courses. To address a common misconception, it is worth clarifying that autodiff is *not* equivalent to symbolic differentiation. While symbolic differentiation is supported by many computer algebra systems (e.g., Mathematica and Maple) and can be a valuable analytical tool in some domains (e.g., mathematical research and education), it faces two challenges for use in the efficient computation of derivatives. Firstly, as the expression for each partial derivative is separate, the size of the program needed to evaluate a full gradient may be exponential in the size of the original function program. Secondly, the function for which derivatives are desired, with whatever complex control flow and side-effect behavior are included in its source language, must first be transformed into a single closed-form mathematical expression, possibly requiring a new form that is exponential in the size of that of the original, before symbolic differentiation can be applied. As is explored in the following subsections, autodiff does not inherently suffer from either of these problems.

Autodiff instead exploits the structure of the considered function, and repeatedly applies local differentiation rules to each nested function. By combining these rules with the standard calculus chain rule, the derivative of $f(\mathbf{x})$ can clearly be written as

$$\frac{df}{dx} = \frac{df_1}{df_2}\frac{df_2}{df_3}\cdots\frac{df_{n-1}}{dx}.$$

As long as each local differentiation rule is precise, this method should carry higher precision than a numerical method and greater memory efficiency than symbolic differentiation. The challenge of implementation is then to find an *efficient* set of operations that can compute this derivative chain.

There are, generally speaking, two different approaches to this challenge, separated by their assumed associativity order of chain rule applications.[101] While the example f given is over real numbers, all domains used in the full system, including vectors, matrices, and quaternions, all support both left and right associativity of chain rule applications, although great care must be taken to not violate the order of underlying non-associative or non-commutative operations.

What is commonly referred to as **forward accumulation** autodiff approaches the problem by associatively grouping these derivative computations in the same order the functions to which they apply would be computed to evaluate the original function. This can be represented by a right-associative order of chain rule applications, where

$$\frac{df}{dx} = \left(\left(\left(\left(\frac{df_1}{df_2}\right)\frac{df_2}{df_3}\right)\dots\right)\frac{df_{n-1}}{dx}\right)$$

means that each local derivative can be simply rewritten as

$$\frac{df_i}{dx} = \frac{df_i}{df_{i+1}}\frac{df_{i+1}}{dx}.$$

Conversely, **reverse accumulation** autodiff approaches the problem by assuming that these operations are left-associative, where

$$\frac{df}{dx} = \left(\frac{df_1}{df_2}\left(\frac{df_2}{df_3}\left(\dots\left(\frac{df_{n-1}}{dx}\right)\right)\right)\right)$$

can similarly be interpreted as leading to

$$\frac{df}{df_i} = \frac{df}{df_{i-1}}\frac{df_{i-1}}{df_i}.$$

[101]Hybrid associativity options are also possible, but are much more complicated and rarely implemented. Previous researchers have shown that such hybrid approaches can be superior in terms of computational and memory efficiency, but finding an optimal hybrid approach is provably NP-complete (Naumann, 2011).

Note that this fundamentally reverses the order of operation of each computation. While f_{i+1} must be evaluated before f_i in the original function, $\frac{df}{df_{i+1}}$ can only be evaluated after $\frac{df}{df_i}$ in reverse accumulation autodiff.

While forward and reverse accumulation might seem similar, a key difference emerges when considering computing multiple partial derivatives for functions with more than one input and/or output. Assuming the differentiation operations have the same types as the original functions, computing the gradient of $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$ with forward accumulation would require m passes, while doing so with reverse accumulation would require n passes. Given that the mathematical tools explored in Section 4 operated over functions with numerous inputs but only one output, one might surmise that this would mean that reverse accumulation would always be faster. However, a single pass of reverse accumulation is usually more expensive than a single pass of forward accumulation, though the difference in computational cost is heavily dependent on the type of function being differentiated. As a result, forward accumulation can be faster than reverse accumulation for scalar valued functions with relatively few inputs, with the tipping point varying based on program type and structure.

An additional important distinction between autodiff approaches rests in the what form the local differentiation operations take. Many autodiff systems specify an alternative set of *interpretive* semantics with which derivatives of a program can be evaluated, the most common form being some form of operator overloading. This contrasts strongly with *transformative* autodiff, which aims to transform the program representing the original expression into another program which can evaluate the derivatives using the standard operational semantics of the language.

While the interpretive autodiff method is often simpler to implement, it suffers from two significant disadvantages. Firstly, none of the static analysis tools specified over the original semantics would operate over these alternative autodiff semantics, severely limiting the efficiency and usability of the resulting differentiation computation. Secondly, these semantics would not facilitate the computation of higher level derivatives, such as a Hessian matrix desirable for some optimization algorithms like Newton's method.

Transformative autodiff solves both of these problems. By expressing the differentiation in the same language and with the same semantics as that of the original program, all of the previously described static analysis tools can operate over this new differentiation program in the same way they would over any other program. Even better, as long as the transformations used produce differentiable programs, transformative autodiff can be recursively applied to compute partial derivatives of any higher order desired.

For these reasons, transformative autodiff is the method used in the current version of the system.[102] Specifically, both forward and reverse accumulation implementations in the current system utilize purely **local** transformations, defined in terms of a given term, its immediate children, and the result of recursively applying the same local transformations to those children.

Moving from the abstract to the concrete, what does locally transformative autodiff look like in this micro-language?

5.4.1 Forward Accumulation

Consider an expression e in this micro-language with scalar independent input variables $x_1, \ldots, x_m$. The goal of transformative forward accumulation autodiff is to find some $\delta_f : t \longrightarrow t$, the output of which can evaluate to the derivative of e relative to *any* single input variable x_i.[103]

Additionally, this *delta* transformation function should preserve the type and order of operations of the original term in the output derivative program. For example, if e is a scalar expression that would evaluate child t_1 before child t_2, then $\delta_f(e)$ must also be a scalar expression that evaluates $\delta_f(t_1)$ before $\delta_f(t_2)$.

Controlling which variable to compute the derivative over is accomplished by modifying

[102]The first version of my system on which Litteneker and Terzopoulos (2017) was based used interpretive forward accumulation autodiff and was, unsurprisingly, substantially slower than the current version for several use cases.

[103]The subscript f for δ_f is used to notationally distinguish the forward accumulation form, δ_f, from the reverse accumulation form, δ_r.

the values assigned to a new set of variables $x'_1, \ldots, x'_m$, which respectively represent the derivative of variables $x_1, \ldots, x_m$. To obtain the derivative of e relative to x_i, one would assign $x'_i = 1$ and $x'_j = 0$ for all $j \neq i$, repeating the process with different choices of i to compute the partial derivatives for many variables.

Formalizing this δ_f function for side effect free expressions in this micro-language follows directly from the standard differentiation rules one might find in any introductory calculus textbook (e.g., (Larson and Edwards, 2009)), and are remarkably straightforward. For conciseness, this notation assumes that all references to a variable with the same identifier x will map under different applications of $\delta_f(x)$ to the same x', regardless of the number of applications or variations in program graph structure.

$$\begin{aligned}
\delta_f(n) &= 0 \\
\delta_f(()) &= () \\
\delta_f(x) &= x' \\
\delta_f(e_1 + e_2) &= \delta_f(e_1) + \delta_f(e_2) \\
\delta_f(e_1 - e_2) &= \delta_f(e_1) - \delta_f(e_2) \\
\delta_f(e_1 \cdot e_2) &= e_1 \cdot \delta_f(e_2) + \delta_f(e_1) \cdot e_2 \\
\delta_f\left(\frac{e_1}{e_2}\right) &= \frac{e_1 \cdot \delta_f(e_2) - \delta_f(e_1) \cdot e_2}{e_2^2} \\
\delta_f(e^n) &= n \cdot e^{n-1} \cdot \delta_f(e) \\
\delta_f(\texttt{peekv}(x)) &= \texttt{peekv}(x') \\
\delta_f(f(e)) &= \frac{f(e + \delta_f(e)) - f(e)}{\delta_f(e)}
\end{aligned}$$

Note that in the case of black box functions there is no other choice but to use numerical differentiation, given the lack of any knowledge as to the body of the function. The particular formulation described above is one simple finite difference method, but may result in numerically instable or undefined values in the cases where the derivative of its input argument, $delta(e)$, has a value near zero. One possible solution to this problem is to simply assume $\delta_f(f(e)) = 0$ if $\delta_f(e) < \epsilon$.

Extending this family of $\delta_f(t)$ rules to control flow statements and sequence expressions is similarly straightforward. The foundational intuition underlying these rules is that the order of evaluation of the differentiation calculation should be equivalent to that of the original program. As a result, any control flow statement s should map to another control flow statement $\delta_f(s)$ that would execute a differentiated version of the same branches as the original s.

$$\begin{aligned}
\delta_f(s : e) =& \delta_f(s) : \delta_f(e) \\
\delta_f(\texttt{skip}) =& \texttt{skip} \\
\delta_f(s_1 \; ; \; s_2) =& (\delta_f(s_1) \; ; \; \delta_f(s_2)) \\
\delta_f(\texttt{while } e_1 < e_2 \texttt{ do } s) =& \texttt{while } e_1 < e_2 \texttt{ do } \delta_f(s) \\
\delta_f(\texttt{if } e_1 < e_2 \texttt{ then } s_1 \texttt{ else } s_2) =& \texttt{if } e_1 < e_2 \texttt{ then } \delta_f(s_1) \texttt{ else } \delta_f(s_2)
\end{aligned}$$

Note that these rules for `if` and `while` statement have two noteworthy issues requiring care. Firstly, side effects that occur during the evaluation of conditions are not factored into the autodiff computation under this formalization. While this is admittedly a bug, I have found it to be satisfactory to simply automatically check whether conditions have side effects and, if detected, display a warning message to alert the user to the issue. Secondly, this autodiff implementation makes no attempt to model any effect changing inputs to the condition expression might have. For example, consider `if y < 0 then x=x`^{2} `else x=1 : x`. Clearly, changing the input value of y can have a significant effect on the value of the expression as a whole, but that effect is difficult to analyze at best and discontinuous (i.e., leading to an undefined derivative) at worst. Fortunately, none of the programs I have experimented with so far depend upon a solution to this problem.

However, including terms with explicit side effects adds an additional level of complexity. If the evaluation of a term t modifies the value of variable x, then $\delta_f(t)$ must modify both x and x' so that the subsequent evaluation of other terms referencing these variables observe an accurate value of both the variable and its derivative corresponding to the original execution of t, with the modification of the derivative variable coming sequentially before the modification of the original variable.

For example, consider the expression $e \coloneqq (y = x - 1); (y = y^2 - 1) : y$. Clearly, the side effects of each assignment term modifying y are integral to the accurate evaluation of subsequent terms referencing y. However, many expressions that reference y transform into derivative expressions that reference both y and y'. If an assignment $y = e_a$ has a reference to y in e_a, as the middle assignment in e does, then any accurate computation of $\delta_f(e_a)$ should utilize the value of y before the original assignment was executed, which means that $\delta_f(y = e_a) :- (y' = \delta_f(e_a); y = e_a)$. Following this rule, as well as those described above, it follows that $\delta_f(e) :- (y' = x' - 0; y = x - 1); (y' = 2yy' - 0; y = y^2 - 1) : y'$. This same idea can be extended to list mutations, leading to the last transformation rules for forward accumulation autodiff below.

$$\begin{aligned}
\delta_f(x = e) =& (x' = \delta_f(e)\ ;\ x = e) \\
\delta_f(\texttt{pushv}(x, e)) =& (\texttt{pushv}(x', \delta_f(e))\ ;\ \texttt{pushv}(x, e)) \\
\delta_f(\texttt{popv}(x)) =& (\texttt{popv}(x')\ ;\ \texttt{popv}(x))
\end{aligned}$$

Note that executing forward accumulation derivative programs created with these rules carries a risk of erroneous side effect duplication, causing incorrect results. In general, this occurs when a term produces side effects that are consumed by something outside the current term.[104] For example, consider computing the derivative of the expression $(y = y + x) : y$ for both x and y with forward accumulation. Two executions of $\delta_f((y = y + 1) : y)$ are needed to evaluate both derivatives, causing the value of y to be incremented twice. To consider an even worse example, $\delta_f(((y = y + 1) : y) * ((y =$

[104]Had I chosen a functional paradigm model, this would not be an issue.

$y + 1) : y)) \coloneqq ((y' = y' + 0); (y = y + 1) : y') * ((y = y + 1) : y) + ((y = y + 1) : y) * ((y' = y' + 0); (y = y + 1) : y')$, incrementing y four times and producing an incorrect derivative. Admittedly, this is another bug of the current formulation, but one that can be automatically detected by checking, for each CFG node p_a generated by the forward accumulation autodiff transformations, whether $\exists p_b$ outside the derivative program such that $dep(p_b, p_a)$ using the dependency query described in Section 5.3.1, and throwing a warning to the user if detected.[105]

Given all of these transformation rules and tools, how efficient is this formulation of forward accumulation? In terms of memory, the necessity of creating a set of derivative variables $x'_1, \ldots, x'_m$ means that the memory complexity of evaluating derivative programs in this formulation is $O(m)$, the same as the original program. However, this ignores the fact that the derivative program must be stored in memory to be evaluated, meaning that the full memory complexity is dependent on the size of the derivative program produced by the transformations.

Many of the local transformations, such as those for addition and control flow statements, produce derivative programs that are $O(n)$ in the original program size. However, some others, such as those for multiplication or assignment, can cause the derivative program to be $O(n^2)$ in the original program size. This can be improved significantly by exploiting the feature that programs in this language are DAGs, rather than trees, allowing a single term that must appear in the derivative program more than once to be reused rather than duplicated. With this reusing trick, the derivative program is guaranteed to be $O(n)$.

However, nowhere is this derivative program size more likely to explode if carelessly implemented than when the derivative of more than one variable is desired. Given an expression e composed of n terms and variables $x_1, \ldots, x_m$, the forward accumulation autodiff formulation described here will produce a derivative program $\delta_f(e)$ with $O(n)$ terms and requiring $O(m)$ new variables. However, how should this $\delta_f(e)$ be used to

[105]This is another case where UD/DU chains, as described in footnote 99, can provide a valuable speedup.

compute the gradient for all m variables?

The default option, mentioned at the beginning of the section, is to execute a single copy of $\delta_f(e)$ m times following different assignments to $x'_1, \ldots, x'_m$.[106] While this ensures that the gradient derivative program is only $O(n)$ in size relative to the original program, it comes with a significant caveat: the flow sensitive simplification strategies described in Section 5.3.3 would be effectively useless, placing an undesirable lower limit on the runtime of the resulting program.

Instead, the system could create m copies of $\delta_f(e)$, one for each partial derivative in the gradient, with each preceded by the appropriate assignments to $x'_1, \ldots, x'_m$ to select the relevant derivative. Running flow sensitive simplification on this would essentially emulate symbolic differentiation, and potentially provide a runtime speedup when evaluating the derivatives. However, this will prove prohibitively expensive for large programs. Not only will the program require $O(mn)$ size, but also the flow sensitive analysis required for simplification may require between quadratic and quartic time in the size of the program analyzed, potentially pushing the computational cost associated with merely forming the derivative program to the terrifying $O(m^4n^4)$ computational complexity. However, this option is still attractive in application domains with relatively small input programs, an opportunity for long computation time during program creation, and a high priority for efficiency when executing derivative programs.

5.4.2 Reverse Accumulation

The goal of transformative reverse accumulation autodiff is to find some $\delta_r : e \longrightarrow s$, which can evaluate the derivatives of e relative to *all* input variables $x_1, \ldots, x_m$ of e.[107] Unfortunately, affecting this in practice presents a complicated and confusing problem.

While this goal might seem syntactically similar to that of forward accumulation, the

[106]Note that this again exploits the DAG structure of the micro-language, and this was actually my principal impetus for choosing a DAG structure over a tree.

[107]The subscript r for δ_r is used to notationally distinguish the reverse accumulation form, δ_r, from the forward accumulation form, δ_f.

distinction between attempting to compute a single scalar derivative and a full gradient as part of a singular program manipulation requires a fundamentally distinct approach. A useful metaphor for understanding this difference is in the direction of information flow. Forward accumulation can be broadly understood as seeding the values of descendant variables corresponding to input derivatives in a differentiated expression, then allowing their seeded values to flow up the program graph in the direction of standard evaluation rules to calculate the derivative. However, in reverse accumulation, the goal is to find a transformation that will allow for the derivative of the expression to be seeded, the value of which can somehow flow opposite to the direction of standard evaluation and into variables corresponding to input derivatives.

The receptacles for this derivative data in reverse accumulating autodiff are partial derivative variables commonly referred to as adjoints, with each expression e and variable x having adjoint variables $\overline{e}$ and $\overline{x}$ respectively.[108] Here, the goal is to find some δ_r such that evaluating $\delta_r(e)$ will result in $\overline{x}$ being equal to $\frac{\delta e}{\delta x}$. At the beginning of the computation, the adjoint for the top level expression is seeded so that $\overline{e} = 1$, then adjoint variables $\overline{e_i}$ for subexpressions e_i are updated based upon their parent terms so that $\overline{e_i} = \frac{\delta e}{\delta e_i}$.

However, effecting the flow of data between adjoint variables is complicated by the possibility of branching and side effects in expressions,[109] as this effectively requires the order of evaluation to be reversed. To accomplish this, a set of stack (i.e., first-in first-out list) data structures are employed that allow for reverse autodiff data to be accessed in the reverse order with which it was saved, a strategy commonly referred to as a "tape or

[108]I have personally always found the commonly accepted "adjoint" name to be somewhat confusing. For real functions with linear derivatives, the reverse accumulating derivatives can be expressed as the adjoint, or transpose, of the forward accumulating derivatives written as a matrix. While this concept fits far less tidily for nonlinear functions, the name adjoint is commonly used in all contexts.

[109]One might ask whether this problem would be simpler if the micro-language was functional, and lacked side effects or imperative-style branching. Unfortunately, I can confidently state that locally transformative reverse autodiff is impossible for functional languages, as transforming a variable usage would require knowledge of other, possibly recursive, variable references. However, either non-local or interpretive reverse autodiff for functional languages may be possible. The only transformative autodiff project for a functional language of which I am aware is Kmett (2022), which uses "benign" side effects to manipulate their tapes.

"Wengert list" for its commonly attributed inventor, Robert Edwin Wengert (Wengert, 1964). In this micro-language, these tapes are used to store both numbers, manipulated with `pushv` and `popv` operations, and statements, manipulated with `pushr` and `execr` operations, with each category serving a similar but distinct purpose. As evaluating derivatives often requires expression values and expressions can have side effects, the values of each intermediate expression and variable assignment must be saved with each evaluation for potential use later, necessitating stacks of numbers. Additionally, branching with `if` and `while` statements can produce sequences of operations that are difficult or impossible to directly reverse, a task easily solved by allowing for stacks of operations to be constructed and evaluated dynamically.

Putting these ideas together into a single set of formal rules is surprisingly complicated, with the full listing of the relevant rules appearing in Appendix A. However, the general strategy for reverse autodiff for an expression e can be summarized as follows.

1. Begin by initializing an empty statement stack x_{rd}. Additionally, empty numeric stacks x_{e_i} and x'_j are initialized as empty for each subexpression e_i and variable x_j.

2. Evaluate a version of the original expression e, where each descendant term has been transformed, that has the following properties.

   - The evaluation of each expression e_i, other than trivial expressions like variable references and number constants, performs the following sequence of operations.

     (a) An operation is pushed to x_{rd} that will pop an element from x_{e_i}, a stack variable for recording the values of the expression, and zero out the subexpression's adjoint variable $\overline{e_i}$.

     (b) The expression is evaluated, and its value is pushed on to x_{e_i}.

     (c) A second operation is pushed to x_{rd} that increases adjoint variables $\overline{e_j}$ for each child expression e_j by $\overline{e_i}\frac{\delta e_i}{\delta e_j}$. To give two examples,

       – the expression $e_1 + e_2$ will produce $\overline{e_1} += \overline{e}$; $\overline{e_2} += \overline{e}$,

  - and the expression $e_1 \cdot e_2$ will produce $\overline{e_1} += \overline{e} \cdot \text{peekv}(x_{e_2})$; $\overline{e_2} += \text{peekv}(x_{e_1}) \cdot \overline{e}$.

  (d) The top value of x_{e_i} is peeked and returned without further modifying the stack.

- Each statement with side effects must push an operation onto x_{rd} such that the program will return to the state present before the statement was evaluated. As all side effects in the micro-language are effected by mutating the value of a variable, this reversibility is generally accomplished by pushing and popping values from a stack x' for each variable x.
- All other terms evaluate normally.

3. Seed $\overline{e} = 1$.

4. Evaluate all statements in x_{rd} in the reverse order in which they were added.

Following the execution of a program following this strategy,[110] each variable adjoint $\overline{x} = \frac{\delta e}{\delta x}$ for every input variable x, with no significant caveats or holes in coverage.

Perhaps the most significant advantage of reverse autodiff is that a single execution can calculate all derivatives with $O(1)$ iterations, as opposed to the $O(m)$ iterations required for m variables with forward autodiff. Additionally, a reverse autodiff program following this strategy has linear size relative to the original size and linear complexity relative to the original complexity for a full gradient, albeit with relatively high constant. However, a significant disadvantage is that the memory complexity is linear with respect to the runtime complexity of the original program. While this issue would prevent the differentiation of long-running expressions, none of the experiments run so far have run into this issue.[111]

[110] To reiterate, the full details are found in Appendix A. Not only are the full rules remarkably complicated, this chapter is already annoyingly long.

[111] Common strategies for overcoming this problem, such as rematerialization and checkpointing, instead store only enough intermediary values to reconstruct the remaining necessary values, although I have not attempted to implement any such strategy in the system.

While reverse autodiff is considerably faster at computing the derivatives relative to many input variables than forward autodiff, derivatives for fewer variables can sometimes be more efficiently computed with forward autodiff. The reverse autodiff transformation requires a considerably greater number of terms and variables, which not only carries a larger computational and memory footprint, but also raises the cost of simplifying any irrelevant portions of the program using the techniques discussed in Section 5.3.3, than forward autodiff. However, precisely characterizing the conditions with which one technique will be more efficient than the other is challenging. For most of the experiments run with my system, reverse autodiff and forward autodiff seem to be roughly comparable in performance with ~ 5 input variables, although different objective functions can cause some variability. As all but a few of the experiments run necessitated optimization over 5 or more variables, reverse accumulating autodiff was much more heavily utilized in my research than forward accumulating autodiff.

5.5 Limitations

As previously mentioned, the full software system has a number of additional features not included in this described micro-language, including linear algebra types, trigonometric operations, I/O functionality with UE4 as well as the file system, and more sophisticated memory management, among other features. However, there are a host of features that both this micro-language and the full system lack, including pointer/reference types, dynamically sized arrays, and functions capable of recursion. While I am aware of nothing intrinsically incompatible between these missing features and the tools outlined here, I have not yet found the need to implement features such as these to support my experiments.

Probably the greatest area for improvement in the current system is in execution speed. Compared to other language systems (e.g., C++, Java, or even Python), the tree-walking interpreter currently used to execute programs is markedly slow, though there are several different approaches that could be taken to improve this. The system could replace the tree-walking interpreter with a bytecode interpreter, requiring less computation/memory overhead for each operation, as well as allowing more of the program to be store in CPU memory. Alternatively, the system could incorporate new translation functionality to take a micro-language program and output corresponding source code in another language (e.g., C++ or Python), allowing for both faster execution with the other language's execution tools and the potential for interoperability. While each of these approaches are enticing, I have not committed any significant effort towards either, as I have found the current speed to be sufficient for the experiments run so far.

However, I would argue the single greatest limitation of the computational systems approach described here rests in the scale of its implementation complexity. While the algorithms, formulas, and judgments interspersed through this section work perfectly well, implementing a system such as this from anything close to scratch represents a significant undertaking. At virtually every phase of development of this project, I dramatically underestimated the magnitude of the engineering tasks that lay ahead, many of which remain challenging even with the roadmap of development this section provides. While I have learned a great deal during my implementation efforts, I do not believe I can, in good conscience, recommend attempting this same route to others. Instead, those wishing to construct a system capable of replicating the experiments in this work would likely find far greater return for their time/energy investment by taking one of the alternative routes suggested beginning on page 111.

CHAPTER 6

Cinematographic Data Collection

One of the goals of my research has been to develop a dataset, with features collected from theatrically released feature films, that is suitable for machine learning or data science tasks in vector-based virtual cinematography. However, this is not a straightforward prospect.

An average feature film is 90-120 minutes[112] at 24-30 frames per second. These $\sim 130 - 200$ thousand frames, each of which contains thousands or millions of pixels, capture a wide diversity of visual material and communicative intent. Images of different actors wearing various costumes in changing poses, captured under varying lighting conditions in disparate types of environments, are used by filmmakers to provoke thought, tears, laughter, and fear in their audiences. As discussed at length in previous sections, each and every film represents an enormity of creative decision making over months of collaborative effort between skilled artists and technicians.

Of course, an audience that was uninvolved in the film's production can see few if any of these decisions directly, instead being only capable of observing the aggregate of a multitude of decisions as visible in the finished film. Considering only one frame, one shot, or one film individually is unlikely to provide enough context to allow for any of these complex decisions to be automatically untangled. By instead considering a large corpus of films, features common among a variety of films, as well as aberrations from common patterns, can be automatically identified.

However, too much data can be problematic. Given the limitations of present technol-

[112]While 90-120 minutes is a standard definition of feature film runtime, the dataset does contain several films that are either shorter or longer than this range, as summarized in Table 6.2.

ogy, a single film contains far too much information to attempt to capture or process all at once. While some combination of computer vision and human annotators could be used to label and describe every element of every frame, doing so could easily exhaust the budget of even a large corporation, let alone a small group of university researchers. To limit the scope of work required to a practicable level, some careful consideration of what criteria the collected data should meet is required.

6.1 Criteria

As a focus for these criteria, it is useful to subtly refine the goal of this dataset as being to facilitate the analysis of vector-based, person-centric cinematographic composition. From this, it seems reasonable to assert that the data collected should answer the following questions:

1. How many people are in each frame?
2. Where are the people in each frame?
3. How large are the people in each frame relative to the frame size?
4. What direction are different parts (e.g., heads, torsos, etc.) of the people facing relative to the camera?
5. How far away are the people from the camera, relative to the other people in the frame?
6. How do all of the above properties change over time for each person in each shot?

Note that, while comparatively complete with respect to the motion of people in the shot, many significant categories of features that could be collected may be omitted as irrelevant to these criteria, including object recognition, action labeling, material reconstruction, and audio analysis, among many others. One feature that is noticeably absent from relevance to these criteria concerns the world-space motion of the camera: a dataset that meets these criteria cannot discriminate between the motion of the camera relative to stationary actors, and the motion of the actors relative to a stationary camera. While I initially intended to include global camera motion in the dataset, my experiments in automatically collecting this data at scale were not successful, as discussed in Section 6.4.

However, two important decisions are left unanswered by these criteria: what films should be analyzed, and on what time scale should data points be collected?

While there was no particularly cogent criteria used in the selection of films, other than a general tendency to pick movies I enjoy and to which I have access, I did try to include representatives of a variety of different genres, styles, and eras, as well as include some films commonly cited in rankings of the best of all time. The dataset contains 13 of *AFI's 100 years...100 movies* ranking (American Film Institute, 1998), and 11 of BFI's top 100 as published in *Sight and Sound* (American Film Institute, 2012). However, there were some categories that were avoided on purely practical grounds. As the goal of this dataset is to collect and utilize person-centric data, the films selected should have clear and recognizable humans in frame. For this reason, I purposefully avoided some titles in animation, fantasy, and science fiction genres, as I found many of the computer vision detectors used in the analysis struggled with non-humanoid and non-photorealistic characters.

Choosing a sample rate for analysis requires even more careful consideration. Analyzing large volumes of frames can quickly grow computationally intractable, forcing a choice of what frames to sample. Given that sequential frames are recorded and played back quickly enough to trick the viewer's brain into perceiving them as fluid motion, many of the available frames will be extraordinarily similar. After some consideration, I decided to balance these considerations by analyzing 2 frames for every second of original footage, or one frame every 0.5 seconds.[113] This data sampling rate keeps the total analysis time near the runtime of the original film, given analysis times listed in table 6.1, allowing for more films to be included in the dataset.

[113]Standard film is recorded and played back at $\sim$ 24 FPS, while television and digital video is usually in the 24-30 FPS range. Sampling at 2 FPS therefore samples 1 frame in every 12-15.

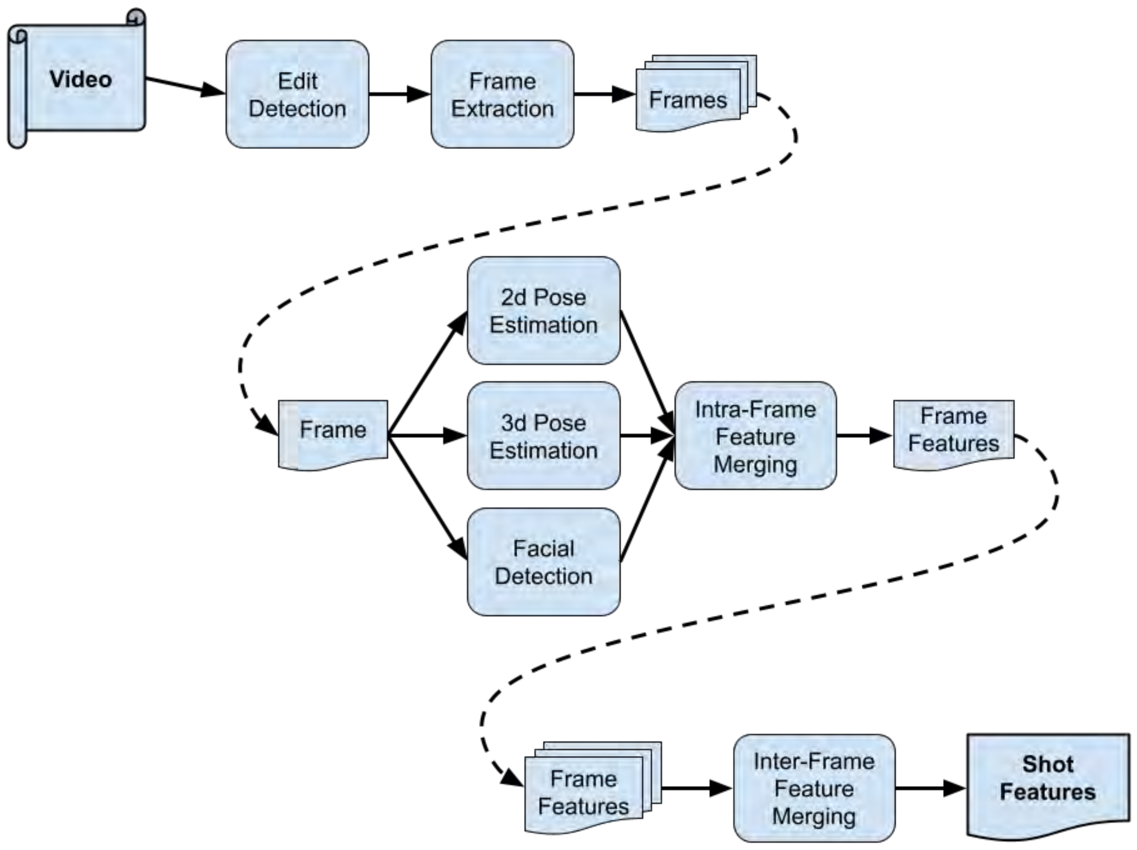


Figure 6.1: Overview of the data collection process.

All told, 60 feature films released over $\sim$ 80 years, comprising $\sim$ 1 million frames across $\sim$ 88K shots, were analyzed. A high level summary of the films analyzed can be seen in Table 6.2, later in this section.

6.2 Data Collection

The data collection process, illustrated in Figure 6.1, can be broadly broken down into three phases:

1. Edits are detected in the full film, and frames are extracted from each continuous shot at 2 FPS.

2. Various features are independently annotated in each frame by a variety of computer

vision detectors, which are then merged together to form a set of data points referencing each detected person in the frame:

   (a) Faces are detected, including facial recognition descriptors.
   (b) Two-dimensional human poses are detected, including estimates of head orientation.
   (c) Three-dimensional human poses are detected, including estimates of body orientation and relative distance from the camera.
   (d) The results of these frame feature detectors are merged.

3. Data from all frames in each shot is merged, so that individual people can be tracked over time in the shot.

For the most part, this data collection process relies heavily upon existing computer vision tools, rather than attempting to build bespoke tools from scratch, and is described in detail in the proceeding subsections. The process as a whole is relatively computationally expensive, although this cost varies significantly depending on the type and content of the film.

One of the most significant speed differences is that high resolution imagery is substantially more expensive than lower resolution imagery. While high definition video[114] allows for higher precision analysis, it takes almost twice as long to analyze an HD video than an SD video with the same number of frames.

A much subtler speed variation is that frames containing more people are computationally more expensive to analyze. While some of this increased cost is intuitively linear in the number of detected people, some phases computations are super-linear in the

[114] Annoyingly, the term "high definition" (HD) video is used by different communities to refer to imagery with more than 480/576/720/1080 rows of pixels, frame rates higher than 50/60 FPS, color spaces with more than 8 bits of depth per channel, or some combination of these qualifiers. I use it here to mean imagery with at least 720 rows of pixels, while standard definition (SD) video has only 480 rows of pixels. Both HD and SD are assumed to play at 24-30 FPS, and no assumptions concerning color depth are made.

Phase	Subphase	SD		HD	
		Speed	Time	Speed	Time
Edit Detection	24 FPS analysis	771.0	3:44	123.5	23:19
	2 FPS export	126.0	2:06	22.4	11:47
	Total	45.3	5:50	7.5	35:06
Face		25.7	10:16	15.2	17:22
2d Pose	Detection	19.9	13:16	20	13:12
	Post Processing	75.0	3:31	48.2	5:29
	Total	15.7	16:47	14.1	18:41
3d Pose		35.3	7:29	21.4	12:20
Frame Merging		10.0	26:24	7.45	35:26
Shot Merging	Shots/second	8.4	2:18	8.4	2:18
	Total	115.1	2:18	115.1	2:18
Total		3.8	69:04	2.2	121:13

Table 6.1: Time estimates required to analyze a single film. This analysis makes several assumptions for simplicity, largely based upon the distribution of data observed in the dataset. The hypothetical film analyzed is assumed to be exactly 120 minutes in length at 24 FPS, for a total of 172,800 total frames. Of those, an average of 2.2 frames per second are assumed to require extraction, following the sampling strategy described in Section 6.2.1, leading to a total of 15,840 frames for analysis. Additionally, The film is assumed to have an average shot length of 13.7 frames per shot, leading to a total of 1,156 shots. Except where otherwise noted, speeds listed are in frames per second, while times are formatted in (minutes):(seconds).

number of people detected per frame or between pairs of frames. As a result, attempting to predict the cost of analyzing a single film can be deceptively difficult.

However, a reasonable set of estimates for the time required to analyze a single film is summarized in table 6.1. The cumulative rate across the dataset was 1.2-4.5 frames analyzed per second, requiring $\sim$ 120 hours of total computing time.[115]

6.2.1 Edit Detection and Frame Extraction

The first step in collecting data from a film is edit detection in an edited video file. While edit detection is a well researched problem, many of the best heuristics and algorithms

[115]This number omits the repeated analyses of a few films while the data collection process was being developed.

are held as proprietary, and are therefore unavailable.

I began my efforts in edit detection with version 0.5.2 of PySceneDetect, an open source edit detection application written in Python (Castellano, 2020). PySceneDetect essentially operates by comparing a subset of pixels in consecutive frames in Hue-Saturation-Luminance (HSL) color space, and triggering a cut detection when the average difference in HSL values between frames raises above a static threshold. While the default cut detection works for many cases, I found its standard method to be somewhat finicky and unreliable for automating edit detection across the diverse films intended to make up the dataset.

To remedy this, I experimented with a variety of different edit detection methods on the dataset released as part of DeepSBD (Hassanien et al., 2017), which includes sequences of frames with $\sim$ 80K hard cuts. In particular, I experimented with delta-thresholding, SVMs, and simple neural networks. For simplicity and speed, I chose the delta-thresholding method, where a cut detection is triggered whenever the average difference between consecutive frames rises at least 20 HSL units higher than the average differences measured over the last 10 frames. This succeeds in correctly detecting more than 99.9% of cuts from DeepSBD's dataset. Note that this method will not detect gradual transitions, such as fades or wipes.

Approximately 7 of the 60 films analyzed were monochrome. Taking differences in HSL space with monochrome pixels is problematic, as any tiny errors in compression or encoding of the video file may emit false cut detections as the HSL jumps significantly from the previous average. To address this, edit detection for monochrome films was performed with only the luminance channel of the HSL color difference.

Once cuts are detected, the frames to be analyzed are extracted from the video and saved to disk. As previously mentioned, I chose to analyze 2 frames from every second of original video. Specifically, the first frame of every shot is always extracted, followed by further frames in the shot at 0.5 second intervals. Additionally, if more than 0.25 seconds are left in the shot after the final frame following this pattern, the last frame in the shot

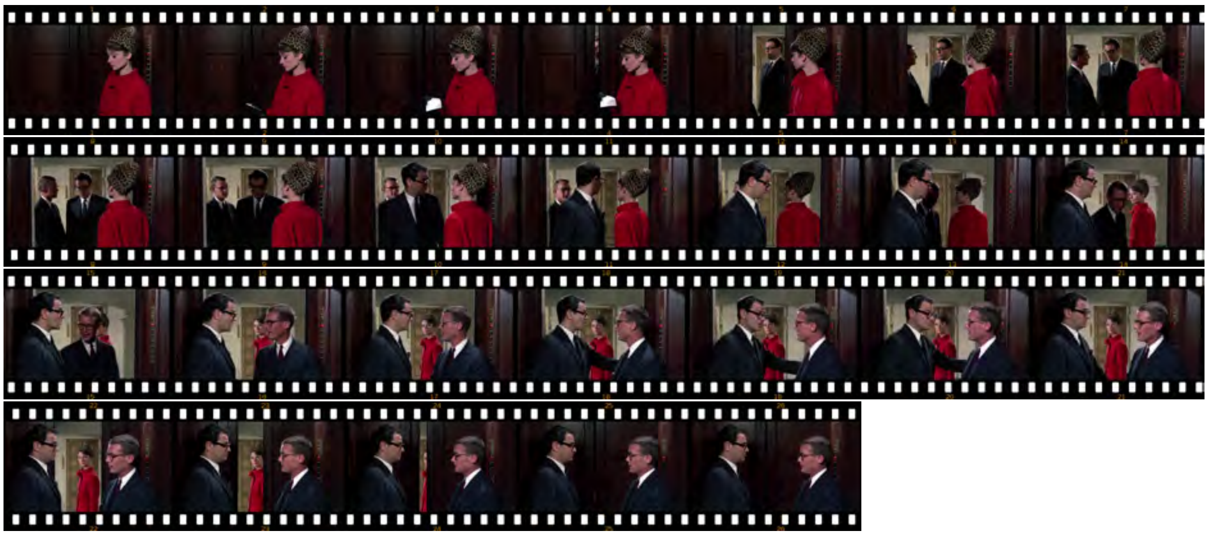


Figure 6.2: An example 2 fps frame sequence for a shot from *Charade*.

is extracted as well.

This strategy was chosen to provide as uniform a representation of each shot as possible, regardless of where on the clock the shot begins or ends, and an example of frames extracted from a single shot can be seen in Figure 6.2. It is worth noting that this strategy results in an average true sampling rate of ~ 2.2 frames per second, slightly in excess of the intended rate.

Even with only the data from this stage, a meaningful pattern emerges. There is a clear correlation between the average length of a shot in a film and the year in which the film was released: films released more recently have tended to have a shorter average shot length (ASL) than films released in the past, as demonstrated in Figure 6.3. The trendline from this roughly aligns with that of other authors, such as (Cutting et al., 2011), suggesting that the edit detection phase of analysis is broadly accurate.

6.2.2 Frame Feature Detection

To detect the desired people related features for each frame, I use a mixture of existing and novel detectors.

There are a variety of issues that can make feature detection with traditional computer

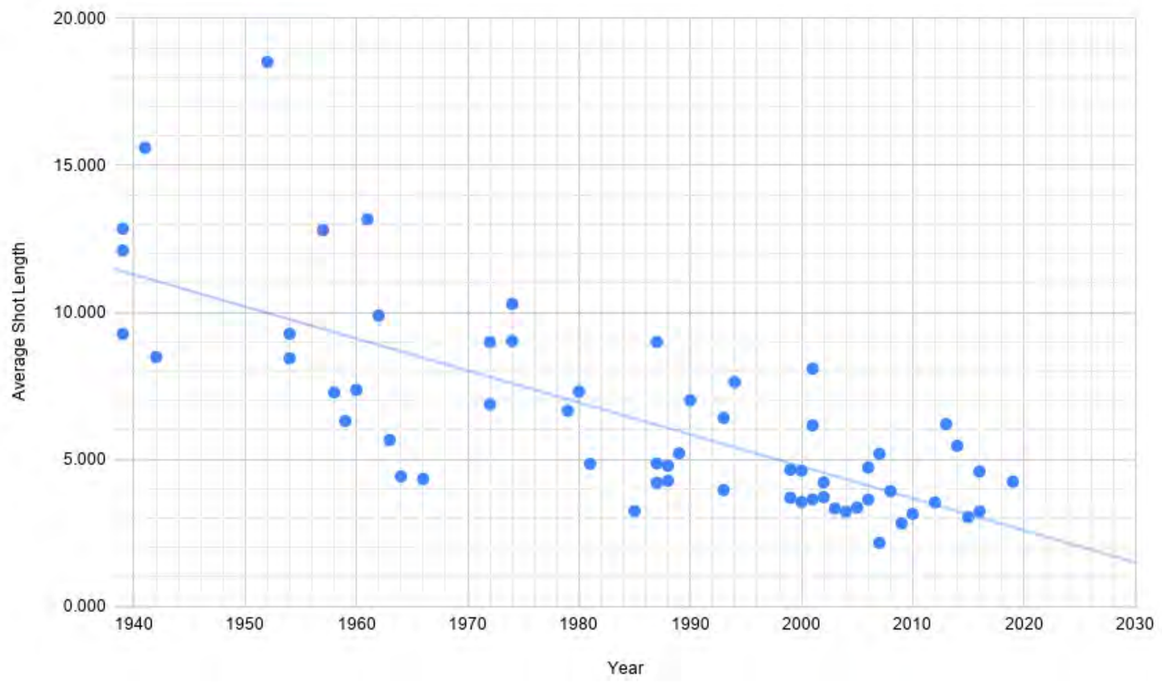


Figure 6.3: Average shot length by year in the dataset.

vision tools difficult for feature films. The images contain frequently occluded views of actors, often out of focus or with poor contrast against the background, in dynamic scenes, and with unknown lens/sensor calibrations (i.e., camera intrinsic matrix). Different detectors deal with different elements of these challenges in different ways, exposing unique strengths and weaknesses in different contexts.

These detectors used here have been selected to detect different parts and features of human persons in frame, with the goal of leveraging relative strengths against others' weaknesses to produce an aggregate that is more reliable and precise than any would be individually.

6.2.2.1 Facial Detection

To detect images of faces in frames in the dataset, I used the python version of Dlib's CNN face detector (King, 2009), which outputs a list of rectangles in image space that the detector believes contain faces. This detector often fails to detect faces if they are smaller

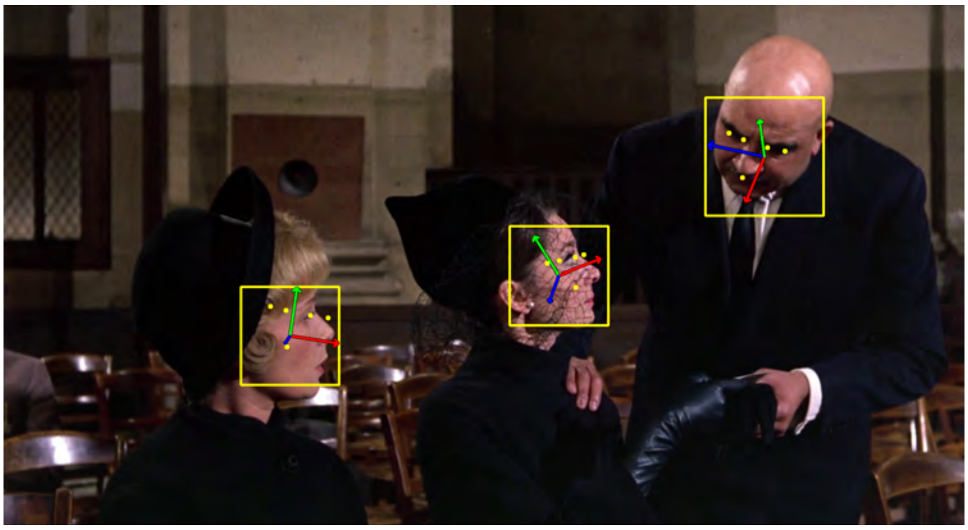


Figure 6.4: Visualization of an example of facial detection results. In this example frame from *Charade*, the detected faces are drawn with yellow rectangles, with estimated orientation drawn as a set of axes and detected facial markers drawn with yellow dots.

than 80x80 pixels, so all frames were upscaled to at least 1 megapixel before detection.

Each detected face in the frame is further analyzed using Dlib's 5 keypoint facial landmark detector, Dlib's face recognition ResNet, and a face orientation estimator called Hopenet (Ruiz et al., 2018).

Dlib's face detector is quite good, rarely triggering on sections of the image that are not faces, but it is not without limitations. As the name implies, it is designed to detect faces. If a person in the frame is near profile, facing away from the camera, or has part of their face occluded or in darkness, the detector is less likely to trigger.

For more general person detection, this face detector can be thought of as having high precision but low recall.

6.2.2.2 2d Pose Estimation

OpenPose is a 2d pose estimation application developed by researchers at CMU (Cao et al., 2019; Simon et al., 2017; Cao et al., 2017; Wei et al., 2016). It detects individual persons in the frame, outputting the image space positions of all detected joint positions from a set of 25 detectable joints for each person detected.

I ran OpenPose on each frame in the dataset. Each detected person in each frame is then further analyzed by a simple feed forward neural network I trained to estimate the size of the person's head in image space, where training data was from the HollywoodHeads dataset (Marin-Jimenez et al., 2011).

This pose head size estimation serves two purposes. Firstly, it provides a basis for comparison with any detected faces as described in 6.2.3. Secondly, it allows for head pose to be estimated without relying on facial detection. However, Hopenet does not particularly work here, as it was designed for estimating orientation of faces and, as such, its valid range of motion only shifts $\sim 90°$in yaw.

Non-frontal head or gaze orientation estimation seems to be in a state of relative infancy. The first significant work I was able to find was Gaze360 from MIT, published and released in 2019 (Kellnhofer et al., 2019). Using their dataset and a modified version of their code, I was able to train a model following their Pinball Static methodology that seemed to behave well with extracted film frames.

While OpenPose is generally a fast and robust detector, it too is not without limitations. Its false positive rate for detection is relatively high, triggering on almost anything person shaped, including stick figures and Japanese characters.[116] Separately, people standing close together can confuse the detector. OpenPose is a bottom-up detector, which first identifies areas of the image likely to contain specific parts of individual people, then groups these parts together into plausible groups forming a connected skeleton. As such, two people standing close together can cause parts to blur together or be assigned to the wrong person.

For general person detection, OpenPose can be thought of as having lower precision but higher recall than the face detector.

[116]One frame in the credits of *Seven Samurai* containing more than a hundred Japanese characters caused OpenPose to crash. OpenPose perceived many of the characters as people and therefore concluded more than 99 people were in the frame, which was greater than its maximum buffer size.

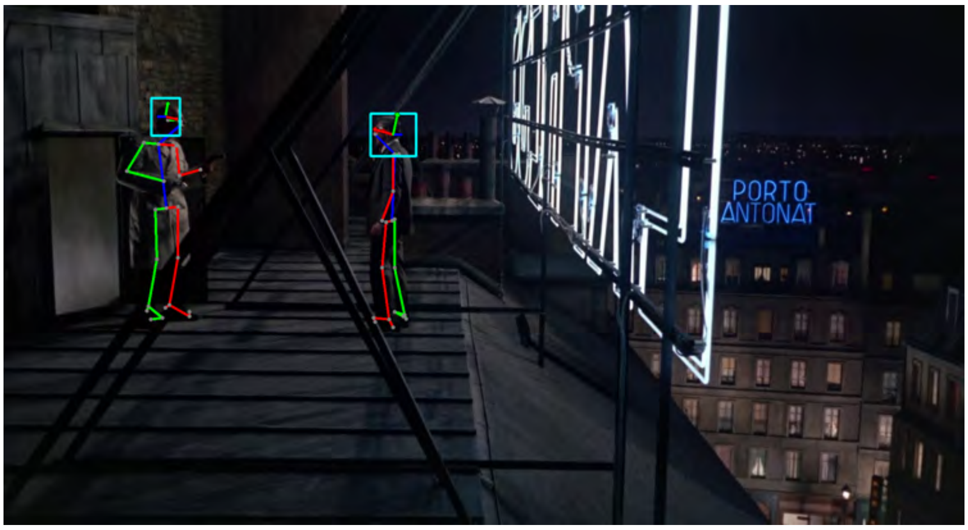


Figure 6.5: Visualization of an example of 2D pose estimation results. The frame visualizes the result of running 2D pose estimation on a frame from *Charade*, in which two characters are standing on a rooftop. Each detected skeleton has the right-hand-side joints of the body colored in green, with the left-hand-side joints colored in red, and central joints (e.g., spine) colored dark blue. Additionally, the estimated head rectangle is drawn in light blue, and the estimated head orientation is drawn as a set of colored axes.

6.2.2.3 3D Pose Estimation

All of the detectors listed so far are essentially 2d. They say nothing about the relative depths or distances between characters, or about the orientation of any part of the body other than the head/face. The most direct way of remedying this would be to utilize a 3d human pose estimator.

While 2D human pose estimation is relatively established, with many functional and well documented systems readily available, monocular RGB 3d human pose estimation appears to still be in its infancy. I tried a number of detectors, including XNect (Mehta et al., 2020), LCRNet (Rogez et al., 2017), and a python project called Lightweight Human Pose Estimation 3d (abbreviated hereafter as LHPE3d) (Osokin and Ageeva, 2020), which is based on (Osokin, 2019). While XNect was very good with full body shots of people, any frame where only part of an actor appeared, either through occlusion or cropping, caused the detector to fail to trigger, a suboptimal feature for collecting data on a variety of shot sizes. LCRNet was better at detecting only partial views of humans, but it is based around a temporal model designed for real time video streams, and therefore tends to struggle when given frames corresponding to 0.5 second intervals. LHPE3d was the only model I experimented with that succeeded in providing estimates for full and partial views of actors over frames at only 2fps.

I ran all frames in the dataset through LHPE3d, and did some comparatively minimal post-processing to extract Euler angles rotations corresponding to the orientation of the head, shoulders, and pelvis of each detected person. It should be noted that LHPE3d outputs both 3d joint position estimations for each detected person, as well as 2d pose estimates of a similar form to that of OpenPose, but across a different skeletal architecture.

While LHPE3d is an excellent piece of software that exceeded all of my requirements, it is not without limitations. Just like OpenPose, LHPE3d is a bottom-up detector, and suffers from some of the same issues when characters stand close together described in the previous section. Additionally, LHPE3d requires a certain number of 2d joints to be detected for its 3d estimator to have any hope of success. If LHPE3d detects only 1 or 2 joints in 2d for a person, it will not attempt to estimate a full 3d skeleton.

It is also worth briefly noting that LHPE3D computes skeletons purely based on cropped portions of the image, then purely translates each skeleton to be merged in to the broader scene. This likely results in somewhat inaccurate rotations between actors or the camera, although it is unclear how much this could be improved without an accurate estimate of the camera intrinsics. However, I have not yet performed any deep analysis into this question.

6.2.3 Intra-Frame Feature Matching

The three detectors described in the previous section operate completely independently of one another. To arrive at a final set of person-centric features for each frame, the results of these independent detectors must be merged. For each person detected by a given detector, the analysis must determine which, if any, persons from the other detectors match.

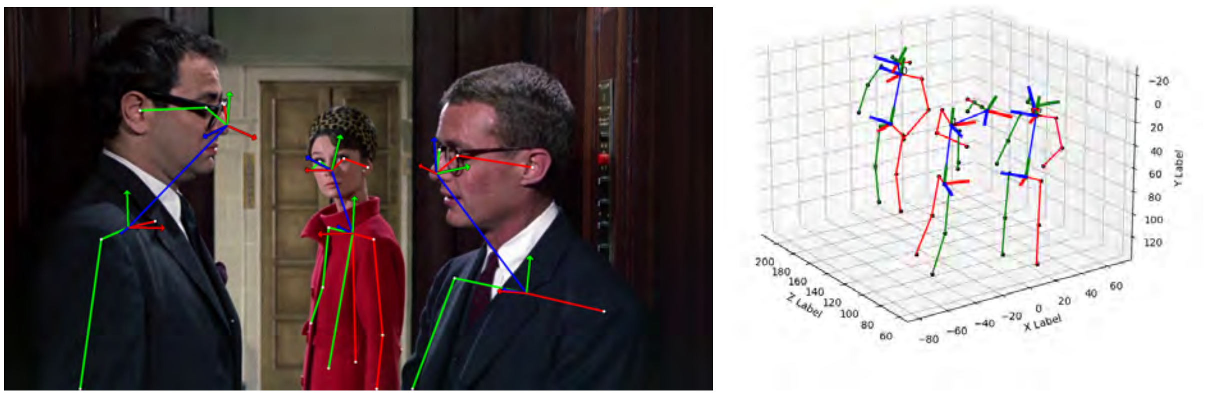


Figure 6.6: Visualization of an example of 3D pose estimation results. The frame on the left shows the results of the 2D joint estimation data from the LHPE3D detector, with green lines corresponding to the right sides of bodies and the red lines corresponding to left sides. The plot on the right shows 3D skeleton estimates, with identical colors. Note that the 3D skeleton includes estimates of joint positions that are not visible in the 2D frame. Both visualizations also show orientation estimates of each actor as a set of axes at the head, shoulders, and pelvis.

Abstractly, this intra-frame matching forms a weighted set cover optimization problem, where the detected results constitute the elements that must be covered and the valid combinations of detector features representing persons from the sets. Unfortunately, finding a globally optimal solution to this problem is NP-hard, and proved too computational expensive for usage on all frames. Instead, I used a greedy approximation algorithm, which is described in algorithm 7. This algorithm operates in $O(n^d log(n^d))$ for a frame where n people were detected by each of d detectors.

However, this algorithm requires a heuristic to evaluate each possible combination of feature results. The heuristic I used took the sum of the following sub-heuristics, each of which considered a pair of detector results.

- Comparing the face with either the 2d or 3d poses is accomplished by considering the bounding box of the face, and the image positions of the pose neck and nose positions. Given a face box with center $\mathbf{c}$ and diagonal $\mathbf{d}$, as well as pose neck position $\mathbf{n}$ and nose position $\mathbf{p}$, the extent to which this face and pose match is expressed by

$$\max\left(0, 0.5 - \frac{||\mathbf{c} - \mathbf{p}||}{||\mathbf{d}||}\right) + \max\left(0, 0.5 - \frac{||\mathbf{c} + 0.5\mathbf{y}\mathbf{d}_y - \mathbf{n}||}{||\mathbf{d}||}\right). \tag{6.1}$$

This same expression can be applied identically to compare detected faces with both 2d and 3d poses.

- Comparing the 2d pose and 3d pose is computed by simply taking the average distance between corresponding points detected by each detector. Specifically, both detectors can output an image space position of each person's nose, neck, pelvis, left shoulder, right shoulder, left hip, and right hip. If each of these N valid corresponding pairs of points is put into a set of tuples P, then this average distance can be simply expressed as

$$\frac{1}{N(1 + \sum_{(\mathbf{p}_i, \mathbf{p}_j) \in P} ||\mathbf{p}_i - \mathbf{p}_j||)}. \tag{6.2}$$

In addition, this heuristic needs some **rejection criteria**. As already noted, the different detectors are likely to detect different numbers of people and, even if each detector detects the same number of people, there is no guarantee that the detected people will be the same in all detectors. If the analysis can determine that a combination is likely to be invalid, then the function should return -1, and the combination will not be considered to be returned by algorithm 7. The simplest rejection criteria I found to be effective was simply comparing the intersection of the various head and keypoint bounding boxes of the detectors. If the 2d pose head bounding box did not intersect the face bounding box, or the bounding box of the 3d pose skeleton keypoints did not intersect with the 2d pose skeleton keypoint bounding boxes, the combination was rejected.[117]

[117]The reader might be wondering about the possibility of rejecting a match between the face detector and the LHPE3D detector alone. I found this to be unnecessary as, when a face was detected, either 2d or 3d pose detectors would fire in the same area with extremely high probability. See table 6.3.

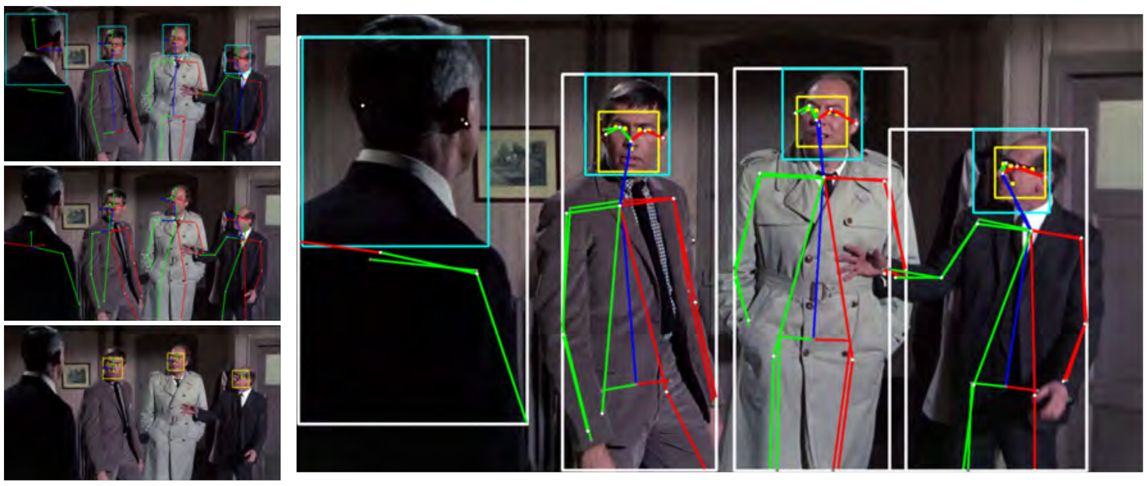


Figure 6.7: Visualization of an example of intra-frame feature matching results. The frame on the right shows the result of intra-frame feature merging with each white rectangle corresponding to a single detected person containing the 2d pose, 3d pose, and facial detection data detected as visualized in the three images on the left.

Algorithm 7: Greedy Detector Feature Matching

```
Given:
    $f(x)$: Heuristic function for scoring a possible match
    $S_1, ..., S_N$: Each $S_i$ corresponds to the set of features detected by detector $i$
1 $A \leftarrow [(s_1, ..., s_N) | s_i \in (S_i \cup \{\text{null}\}) \forall i \leq N]$
2 $R \leftarrow \{\}$
3 foreach $a \in A$ in descending order by $f(a)$ do
4     if $f(a) \geq 0$ and no element of $a$ is already represented in $R$ and
       $a$ contains at least one element that is not null then
5         $R \leftarrow R \cup \{a\}$
6 return $R$
```

6.2.4 Inter-Frame Feature Matching

Lastly, in order to capture the motion of people as a shot progresses, the data from consecutive frames in the same shot must be merged.

This inter-frame problem has many similarities to the intra-frame matching problem in the previous section. We have sets of people in consecutive frames, and the goal is to identify which detected people in some frame correspond to which detected people, if any, in the succeeding frame. This can be formalized as exactly the same sort of weighted set cover problem, and I utilized the same method, algorithm 7, to solve it.

Of course, using this algorithm requires the definition of another heuristic to score each prospective match. This heuristic simply sums similarity scores between alike feature types between frames, as follows:

- If considering a match where both frames contain a detected face, the 128d facial recognition vector provides an excellent means of comparison. The documentation of the model states that any two faces whose Euclidean distance is less than 0.6 should correspond to images of the same person. With $\mathbf{f}_a$ and $\mathbf{f}_b$ corresponding to the facial recognition vector of the two frames being considered, I found

  $$\frac{0.6}{\epsilon + ||\mathbf{f}_a - \mathbf{f}_b||} \tag{6.3}$$

  modeled matching with reasonable accuracy. Note that this does not reject matches where $||\mathbf{f}_a - \mathbf{f}_b|| \geq 0.6$, as I found that a single actor could break this limit with certain actions, such as the donning or removal of a hat or glasses.

- If considering a match where both frames contain a 2d or 3d pose, motivating a heuristic is difficult. At 2fps, an actor's motion in image space can vary tremendously from frame to frame. An actor that is unlikely to move more than 10% of the way across the frame in 0.5 seconds may cross from one side of the screen to the other in a close up. Furthermore, an actor detected in one frame can be missing in the next, either because they have left the camera's field of view or been temporarily occluded by another object in the scene.

  Absent a better idea, I chose instead to take inspiration from a manual examination of a few shots in the dataset, and observed that actors tended to remain the same size in the frame during even the largest and most sudden of motions. I modeled this behavior as

  $$\frac{1}{N} \sum_{\mathbf{l} \in L} \frac{\min(\mathbf{l}_a, \mathbf{l}_b)}{\max(\mathbf{l}_a, \mathbf{l}_b)}, \tag{6.4}$$

  where L is the set of N 2d skeleton edge lengths which are detected in both frames.

This same heuristic is applied to both the 2d poses detected by OpenPose, as well as the 2d data output by the 3d pose detector LHPE3d.

- In the event that the no alike-feature detector is available (e.g., the person in the preceding frame only contains a face, and the person in the succeeding frame only contains a 2d pose), I fall back to simply computing the intersection-over-union, or Jaccard index, of the bounding box containing all of the detected features for each person, $\frac{\text{area}(A \cap B)}{\text{area}(A \cup B)}$ where A and B are the bounding boxes for the preceding and succeeding frames, respectively.

While this heuristic is satisfactory for many of the shots I have manually examined, such as the shot demonstrated in Figure 6.8, it is far from perfect, failing most frequently when actors become occluded or make large, sudden motions. The probability of such a failure seems to increase as the composition becomes tighter, with an actor's abrupt twitch in an extreme-close-up potentially triggering their detection as a new character. Additionally, any scenario in which a character is not detected for one or more frames before being detected again will trigger their being detected as two different people.

6.3 Dataset at a Glance

The dataset captures information from over 1M frames, comprising an estimated $\sim$ 88K shots, from 60 feature films released between 1939 and 2019. A full inventory of all films analyzed, as well as some basic statistics of the data collected from each film, is listed in Table 6.2.

However, because the dataset is person-centric, different frames have different amounts of detected data. Only $\sim$ 871K frames, or 86.8%, contain any detected people at all, although the rate of detection varies significantly between films. Much of this variation is likely the result of some films focusing more on non-human subjects than others. For example, only 74% of the dinosaur heavy *Jurassic Park*'s frames contain people,[118], while

[118]According to (Effron and Gowen, 2018), on screen dinosaurs appear in only $\sim$ 15 of the film's $\sim$ 127

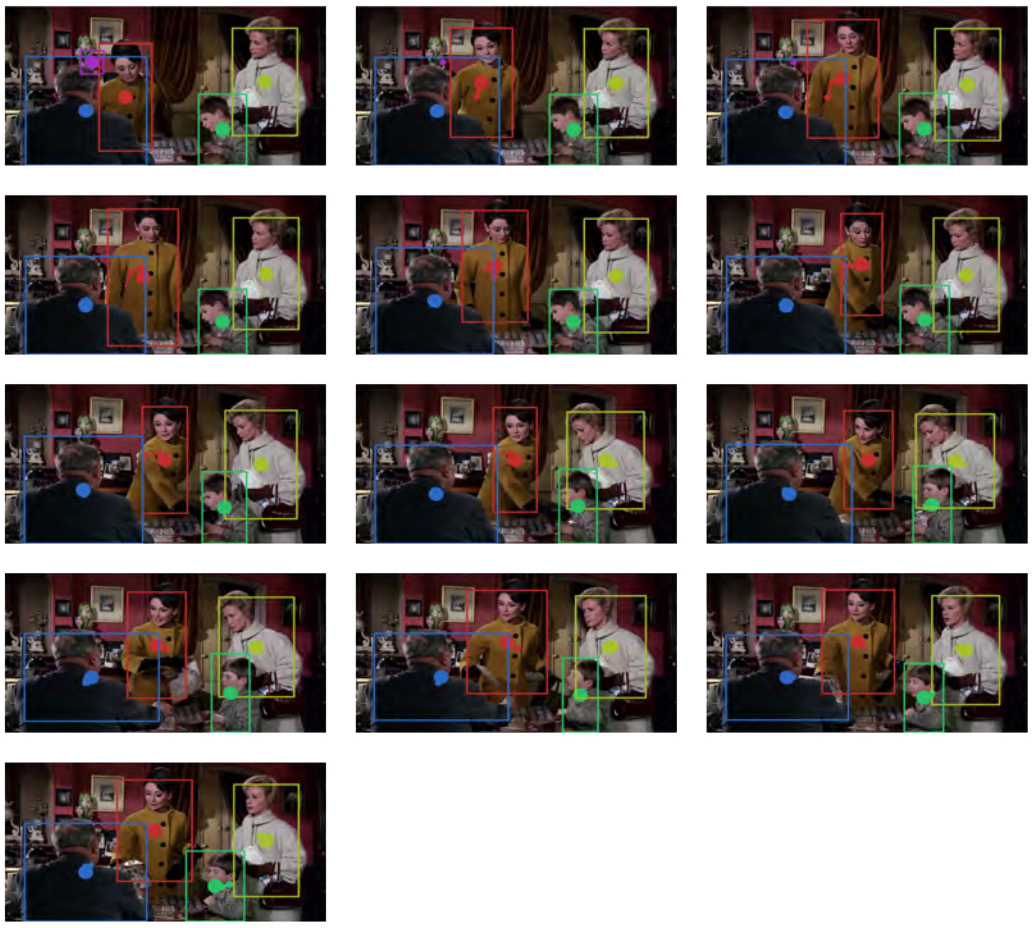


Figure 6.8: Example of successful of inter-frame merging results. The frames are all from a shot in the film *Charade*. Each rectangle in each frame corresponds to the bounds of a tracked person, with the color of each person's rectangle remaining consistent across all frames. To help visualize the motion of each actor, each frame also shows the path that the center of that person's rectangle has taken over the last 5 frames with a correspondingly colored line. Note that in this shot, the tracking correctly tracks the four people in the shot throughout the scene.

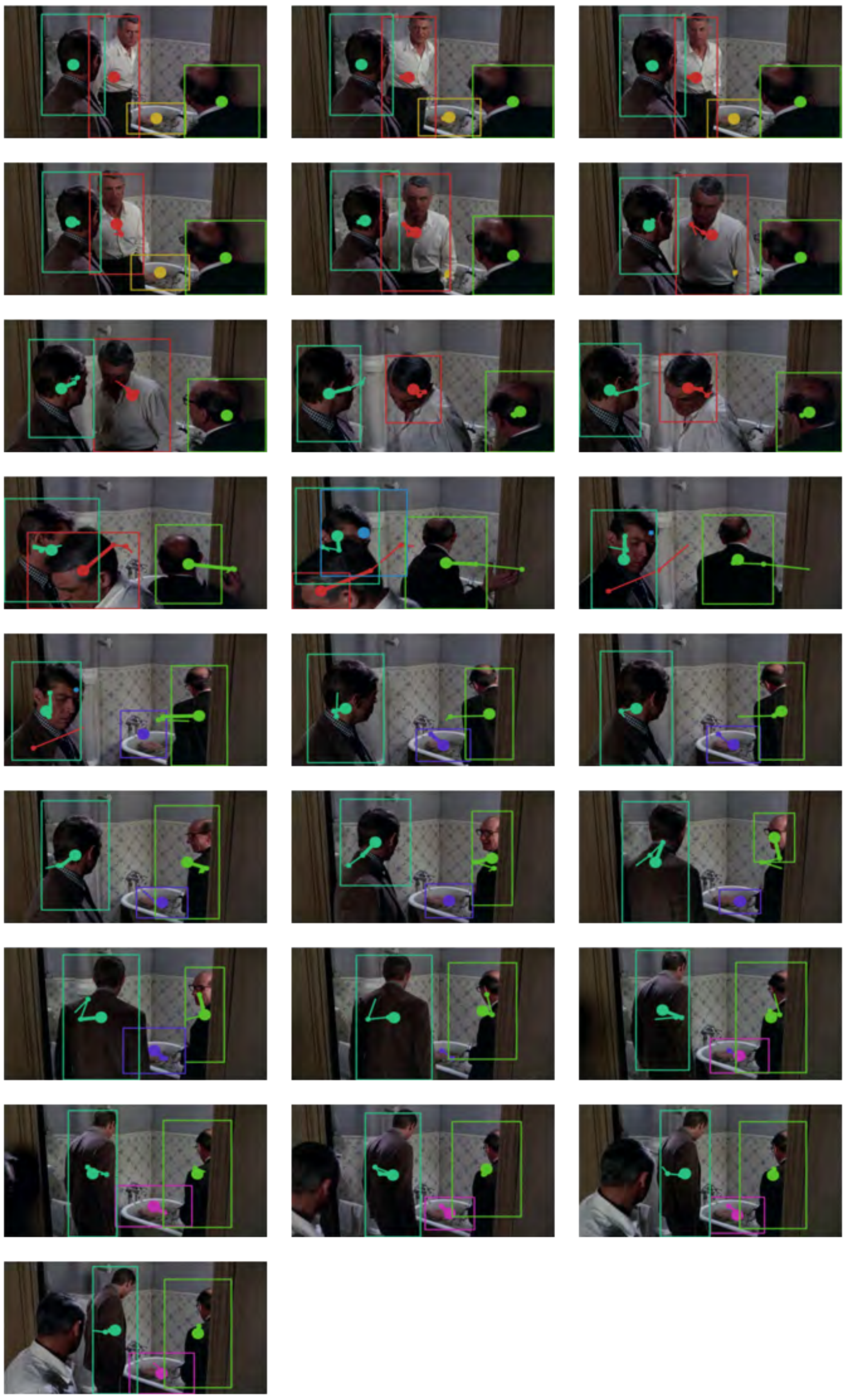


Figure 6.9: Example of less successful inter-frame merging results. This figure follows the same format as Figure 6.8, with different colors corresponding to the detection of different people across the frames. However, the visualized shot from *Charade* has several noticeable merging failures, with the character in the bathtub being detected as three different people over the course of the shot after occlusions for one or more frames.

Title	Year	Director	Duration	Shots	Frames	Frmes w/ People	Total People
A Funny Thing Happened On The Way To The Forum	1966	Richard Lester	1:37:12	1,346	13,063	12,390	44,871
Airplane!	1980	Zucker, Abrahams and Zucker	1:27:42	720	11,276	9,781	32,967
Argo	2012	Ben Affleck	2:00:23	2,040	16,554	13,600	36,267
Bourne Identity	2002	Doug Liman	1:58:18	1,683	15,935	12,185	21,914
Casablanca	1942	Michael Curtiz	1:42:38	726	13,070	12,520	45,361
Casino Royale	2006	Martin Campbell	2:24:11	2,380	19,790	16,451	56,341
Charade	1963	Stanley Donen	1:53:36	1,205	14,879	13,801	36,634
Citizen Kane	1941	Orson Welles	1:59:23	459	14,786	13,255	48,817
Die Hard	1988	John McTiernan	2:12:08	1,857	17,758	14,605	34,335
Goldfinger	1964	Guy Hamilton	1:50:01	1,493	14,746	11,905	30,568
Gone With The Wind	1939	Victor Fleming	3:53:08	1,120	29,129	25,635	66,696
Gosford Park	2001	Robert Altman	2:17:01	1,017	17,471	16,220	46,043
Hero	2002	Zhang Yimou	1:39:09	1,602	13,546	10,690	27,324
Hot Fuzz	2007	Edgar Wright	2:00:53	3,356	17,976	15,032	34,298
Indiana Jones and the Last Crusade	1989	Steven Spielberg	2:06:53	1,462	16,738	13,851	43,275
Inside Man	2006	Spike Lee	2:08:37	1,633	17,119	14,709	44,429
Iron Man	2008	Jon Favreau	2:06:01	1,929	17,136	13,478	35,159
Jurassic Park	1993	Steven Spielberg	2:06:28	1,184	16,395	12,214	28,756
Knives Out	2019	Rian Johnson	2:10:06	1,841	17,513	15,584	33,619
Lawrence of Arabia	1962	David Lean	3:47:00	1,377	28,650	24,638	117,502
Lethal Weapon	1987	Richard Donner	1:49:33	1,352	14,544	12,162	26,137
Life of Brian	1979	Terry Jones	1:33:45	845	12,138	11,063	43,803
Midnight Run	1988	Martin Brest	2:06:19	1,585	16,816	15,305	45,177
Midnight Special	2016	Jeff Nichols	1:51:57	1,463	14,935	11,714	26,453
North By Northwest	1959	Alfred Hitchcock	2:16:25	1,298	17,702	16,133	56,884
Ocean's 11	2001	Steven Soderbergh	1:56:34	1,136	15,154	13,137	46,839
Pirates Of The Caribbean The Curse Of The Black Pearl	2003	Gore Verbinski	2:23:12	2,583	19,842	15,874	39,194
Police Story	1985	Jackie Chan	1:40:28	1,860	13,993	12,943	38,243
Psycho	1960	Alfred Hitchcock	1:48:59	888	13,982	12,079	22,469
Raiders Of The Lost Ark	1981	Steven Spielberg	1:55:19	1,427	15,322	13,063	52,223
Rear Window	1954	Alfred Hitchcock	1:52:33	800	14,330	13,087	23,022
Seven Samurai	1954	Akira Kurosawa	3:21:46	1,306	25,451	23,458	99,787
Shanghai Noon	2000	Tom Dey	1:50:26	1,867	15,168	13,117	30,489
Shawshak Redemption	1994	Frank Darabont	2:22:33	1,121	18,277	15,758	56,641
Sherlock Holmes	2009	Guy Ritchie	2:08:25	2,728	18,242	15,282	43,606
Singin In The Rain	1952	Stanley Donen	1:42:46	333	12,672	12,254	49,921
Snatch	2000	Guy Ritchie	1:42:43	1,336	13,714	12,061	32,791
Snowpiercer	2013	Bong Joon-ho	2:06:12	1,221	16,386	12,522	33,874
The Fugitive	1993	Andrew Davis	2:10:16	1,976	17,684	15,043	39,730
The Godfather	1972	Francis Ford Coppola	2:57:09	1,182	22,454	20,311	72,453
The Godfather Part 2	1974	Francis Ford Coppola	3:22:06	1,343	25,618	23,875	108,839
The Grand Budapest Hotel	2014	Wes Anderson	1:39:56	1,098	13,116	10,871	32,137
The Green Mile	1999	Frank Darabont	3:08:38	2,436	25,142	21,434	43,308
The Hunt For Red October	1990	John McTiernan	2:15:08	1,156	17,397	14,230	31,855
The Incredibles	2004	Brad Bird	1:55:25	2,153	16,046	11,366	19,135
The Last Emperor	1987	Bernardo Bertolucci	3:38:36	1,458	27,732	24,817	91,715
The Lord Of The Rings The Fellowship Of The Ring	2001	Peter Jackson	2:58:25	2,940	20,033	14,457	24,417
The Man from Nowhere	2010	Lee Jeong-beom	1:59:19	2,276	16,656	13,135	26,589
The Man From Uncle	2015	Guy Ritchie	1:56:30	2,362	16,417	13,223	26,847
The Matrix	1999	The Wachowskis	2:16:18	2,215	18,622	14,639	26,954
The Nice Guys	2016	Shane Black	1:55:57	2,156	16,127	14,291	32,397
The Princess Bride	1987	Rob Reiner	1:38:11	1,402	10,824	9,712	20,610
The Wizard of Oz	1939	Victor Fleming	1:41:48	659	12,888	11,511	42,570
Throne Of Blood	1957	Akira Kurosawa	1:49:40	514	13,660	11,758	30,936
V For Vendetta	2005	James McTeigue	2:08:34	2,295	18,268	14,819	28,331
Vertigo	1958	Alfred Hitchcock	2:12:31	1,093	16,559	14,733	29,861
What's Up, Doc?	1972	Peter Bogdanovich	1:33:50	819	12,099	10,902	44,781
Yojimbo	1961	Akira Kurosawa	1:50:50	505	13,769	12,813	49,388
Young Frankenstein	1974	Mel Brooks	1:45:38	616	13,318	11,829	39,196
Zodiac	2007	David Fincher	2:37:40	1,824	20,781	18,049	43,259
Totals			127:45:57	88,711	1,004,345	871,374	2,538,037
Averages			2:07:46	1,479	16,739	14,285	41,607

Table 6.2: Inventory of Films Analyzed

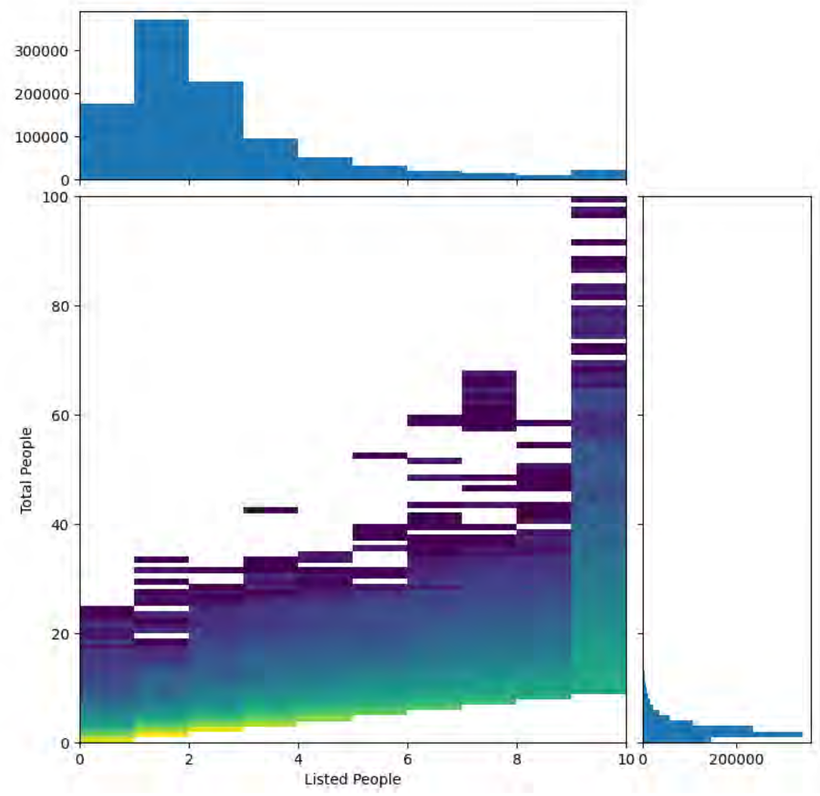


Figure 6.10: Distribution of number of people detected in each frame. The vertical "Total People" axis corresponds to the total number of detected people in each frame. The horizontal "Listed People" axis corresponds to the number of people that would be listed in a frame vector, with data after inter-frame feature matching, with components for 10 or fewer people.

Casablanca has 96% of its frames containing people. Only $\sim$ 6.3K of the $\sim$ 88K shots, $\sim$ 7%, of the total shots did not contain any detected people in any frames.

Diving slightly deeper into this distribution, non-empty frames have differing numbers of detected people, as visualized in Figure 6.10. The number of detected people ranges from an obvious minimum of 0 to a maximum of 128,[119] with a mode and median of 1 detected person representing 34% of all frames. However, 97% of frames have 10 or fewer

minutes. In other words, *Jurassic Park* is $\sim$ 12% dinosaurs. Interestingly, assuming the dinosaurs and humans appear in non-overlapping sets of frames, people frames and dinosaur frames would total to $\sim$ 86%, close to the average of all films.

[119]While surprising, this appears to be relatively accurate, and corresponds to a wide shot of a busy room in shot 2321 of *Sherlock Holmes*.

Face	2d Pose	3d Pose	People	Percent
✓	✓	✓	794207	31.3%
✓	✓		6741	0.3%
✓		✓	5050	0.2%
✓			4807	0.2%
	✓	✓	1219134	48.0%
	✓		289355	11.4%
		✓	218743	8.6%

Table 6.3: Summary of Detections in the Dataset by Detector Type

people.

Furthermore, different detected people have varying sets of matched information detected, as broken down in the following table. While only 32% of the $\sim$ 2.5M detected people have detected faces, 91% have matching 2d poses and 88% of people have 3d poses. The 2d and 3d pose detectors co-occur for 79% of detected people, which suggests a relatively high level of agreement between those detectors.

6.4 Limitations

While the dataset met all of my requirements, it is far from perfect. As already noted, there are a number of different limitations and common failure cases in the intra and inter frame matching operations, as well as the other detectors and estimators used.

The most significant limitation of this dataset is that it is egocentric: it is impossible to differentiate between motion of the actors and motion of the camera. I attempted a variety of commercial and open-source software[120] to track the motion of the camera independent of any actor motion at either 2 FPS or the higher native framerate, without success. Unfortunately, the uncalibrated, dynamic, and out of focus nature of the backgrounds in the majority of the shots in this dataset make them incredibly poor candidates for existing matchmove, 3d reconstruction, or VSLAM automation software solutions. While a skilled human VFX technician might be able to achieve satisfactory results by manually tuning

[120]Including SynthEyes, PFtrack, 3dEqualizer, Adobe After Effects, Blender, and MeshRoom.

software parameters for each shot, even spending an average of a single minute per shot would require $\sim 1,479$ hours of human labor for the $\sim$ 88K shots in the dataset. Perhaps a middle ground where something like optical flow was used as the basis for a coarse estimation of the camera motion type (e.g., pan and/or dolly) might make a beneficial addition to the dataset, but I have not attempted to incorporate any such features so far.

Additionally, this dataset lacks an enormous amount of non-human data present in the video, including scene layout, costumes, lighting, audio, and more. However, none of these features are relevant for the purposes of either analytical or generative vector-based virtual cinematography.

I also ran several failed experiments attempting to better utilize the extracted facial recognition data. The most interesting, at least to me, was an effort to count the number of actors in a film by clustering faces with the facial recognition data.[121] The Dlib documentation asserts that any two faces whose facial descriptors have a Euclidean distance less than 0.6 should represent the same face. However, when I tried running an OPTICS clustering algorithm on the points with 0.6 as the EPS across the $\sim$ 10K faces detected in a single film, the algorithm collapsed all the faces into a single cluster. The only conjectural explanation I could reach is that the number of faces was so large, and under such a widely varying number of viewpoints, lighting conditions, facial expressions, etc. that the probability of overlap approaches 1. Reducing the EPS to 0.3-0.4 can vary the number of detected clusters to a more reasonable number in the tens or hundreds, but doing so frequently separated detections of the same actor's face into different clusters. Perhaps some flavor of random sample consensus between faces in different shots might produce better results, but I have not attempted this.

[121]I briefly held an ambition to use such data to automatically generate diagrams of actor interactions over the course of a film, somewhat like the work of Padia et al. (2019) and the XKCD artwork that inspired their work.

CHAPTER 7

Manually Defined Cinematographic Objective Functions

While any valid objective function can be input by the user, the system supports common cinematic and photographic compositional aesthetics by providing a predefined set of novel objective functions. These mimic the types of shot descriptions commonly used on "shot lists"[122] in the preproduction of a film by live action directors and cinematographers. Additionally, several recipes are provided with which common cinematographic compositions can be easily achieved as a combination of these objectives.

7.1 Data Inspiration

Even before attempting any complicated data science or machine learning, there are some interesting observations that can be made using simple manual data analysis.

We can ask whether there are any significant correlations that might expose either biases in the detectors or evidence of cinematographic patterns. Figure 7.1 shows how various properties of the head rectangles estimated from 2d pose information for each detected person, where all positions are in a normalized display coordinate system.

As visible from this diagram, persons can be detected anywhere in the frame, with head centers possible anywhere inside the bounds of the frame. However, heads are most concentrated in a roughly rectangular region where $-0.6 \leq x \leq 0.6$ and $0 \leq y \leq 0.6$. A very close examination of this region in the center-x/center-y plot shows an interesting

[122]See Section 2.3.

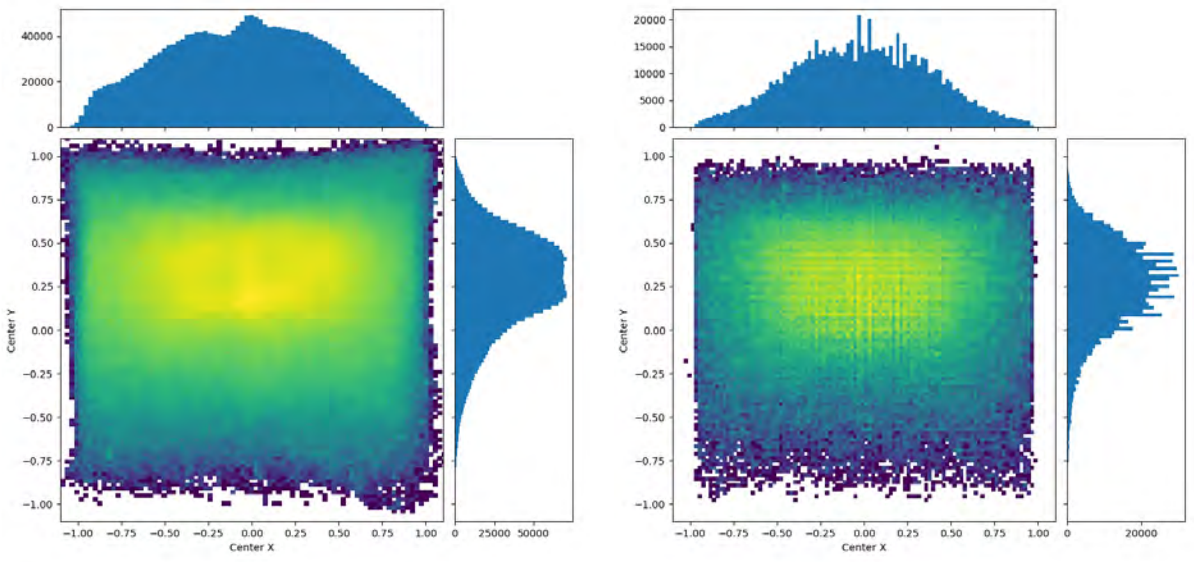
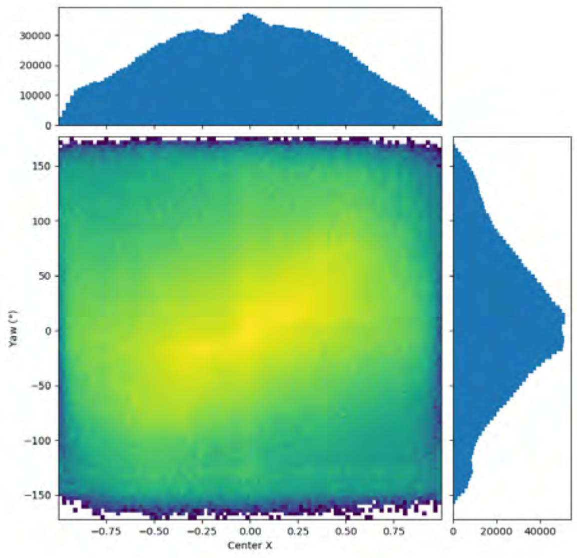


Figure 7.1: Histograms of centers of heads estimated from detected 2d person poses. The plot on the left shows heads detected with the 2D pose detector, and the right plot shows heads detected with the face detector.

double holed pattern. The left and right bars of this pattern center at $x \approx \pm 0.33$, which supports the existence of a horizontal rule of thirds as a cinematographic rule for positioning heads. However, the top and bottom bars of this pattern center around $y \approx 0.15$ and $y \approx 0.5$, suggesting that the vertical rule of thirds does not match the behavior modeled in 7.2.5.

Additionally, a limitation of the nature of the analysis can be observed in the center-x/width and height/center-y plots: large heads can only be detected when there is sufficient separation from the edge of the frame. Furthermore, the width and heights of heads are highly linearly correlated, having a Pearson correlation coefficient of 0.93.

There is also a noticeable amount of correlation between the center of heads and the estimated yaw, although the Pearson correlation coefficient is at a low 0.18. While this correlation could be interpreted as supporting look space in the dataset, a stronger interpretation is that characters on the left side of the frame are more likely to be looking right, and characters on the right side are more likely to be looking left.



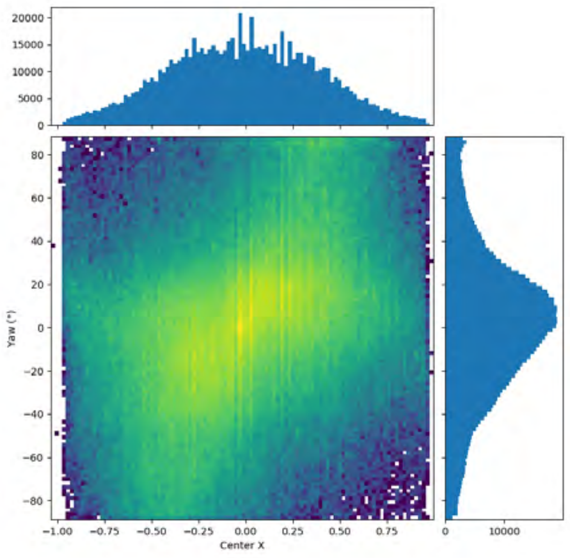


Figure 7.2: Histograms of head centers relative to estimated yaw. The plot on the left shows heads detected with the 2D pose detector, and the right plot shows heads detected with the face detector.

7.2 A Library of Objectives

While the components of this system that are integrated with *Unreal Engine* rely on its particular coordinate system (left-handed, z-up, centimeter unit scale, etc.), the functions listed here are written to be as generic as possible. As several rules require the projection of a point onto the image plane, the function $\Pi(\mathbb{R}^3) \to \mathbb{R}^2$ is used to denote such a projection from world space to screen space, where any point $\mathbf{p}$ that will appear in frame will have $|v| \leq 1, \forall v \in \Pi(\mathbf{p})$. Note that this projection function must have knowledge of the current camera parameters (e.g., position, orientation, field of view, etc.), which are not explicitly listed as arguments for many functions.

All of the functions described below are point-based, meaning that all details of the scene are collected in the form of observing the position, orientation, etc., of specific points connected to objects of interest in the scene. This requires that complex elements in the scene be represented by multiple points. This point-based model varies from the explicit geometric models of (Bares et al., 2000a; Jardillier and Languénou, 1998), where each camera pose is evaluated based on observations of geometric primitives approximating complex objects, as well as from the rendering-based methods of (Burelli et al., 2008; Abdullah et al., 2011), where each camera pose is evaluated by analyzing a full rendering of the scene from that pose.

7.2.1 Frame Position

One of the simplest features a user might want to control is the position that an object should have in the image. To support a wider range of specifications, several objective functions are predefined for different types of desired frame position behavior.

At the simplest level, a user might simply want some world space point $\mathbf{p}$ to be as close as possible to some desired image space point $\mathbf{a}$, which can be simply modeled as

$$f_{point}(\mathbf{p}, \mathbf{a}) = ||\mathbf{p} - \mathbf{a}||^2.$$

However, this point model is overly simplistic for many applications. Frequently, users simply desire that a particular point be positioned within some region on screen, where any position within that region is equally desirable. Here are two predefined objective functions for modeling either an elliptical,

$$f_{ellipse}(\mathbf{p}, \mathbf{a}, \mathbf{d}) = \max(d_x d_y \sum_{i \in \{x,y\}} (\frac{\Pi(\mathbf{p})_i - a_i}{d_i})^2 - 1, 0),$$

or rectangular,

$$f_{rectangle}(\mathbf{p}, \mathbf{a}, \mathbf{d}) = \sum_{i \in \{x,y\}} \max(\text{abs}(\Pi(\mathbf{p})_i - a_i) - d_i)^2,$$

desirable frame region with center position $\mathbf{a}$ and half-size dimensions $\mathbf{d}$. Both are plotted in Figure 7.3.

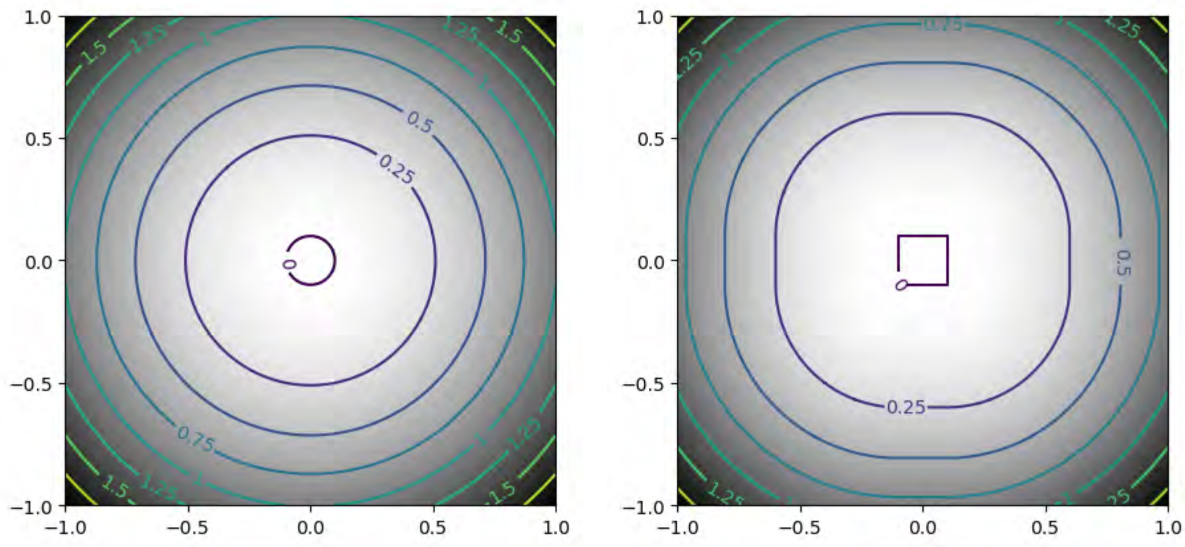


Figure 7.3: Contour plots of frame positioning functions $f_{ellipse}$ (left) and $f_{rectangle}$ (right).

7.2.2 Visibility

An object the user desires to be visible must appear on-screen, but this apparently simple behavior requires two different types of objectives, denoted *frame bounds* and *occlusion/collision*, both of which must be satisfied for an object to be visible.

7.2.2.1 Frame Bounds

The frame bounds rule is simply that any point that the user desires to be visible must appear inside the bounds of the frame. Modeling this is fairly simple using projective geometry.

Given world space point $\mathbf{p}$ and constants $x_{\text{low}} < x_{\text{high}}$, where $-x_{\text{low}} = x_{\text{high}} = 1$ ensures a point is simply in frame and $0 < (|x_{\text{low}}|, x_{\text{high}}) < 1$ will produce headroom, the frame-bounds rule can be evaluated as

$$f(\mathbf{p}) = \sum_{x \in \Pi(\mathbf{p})} g(x), \tag{7.1}$$

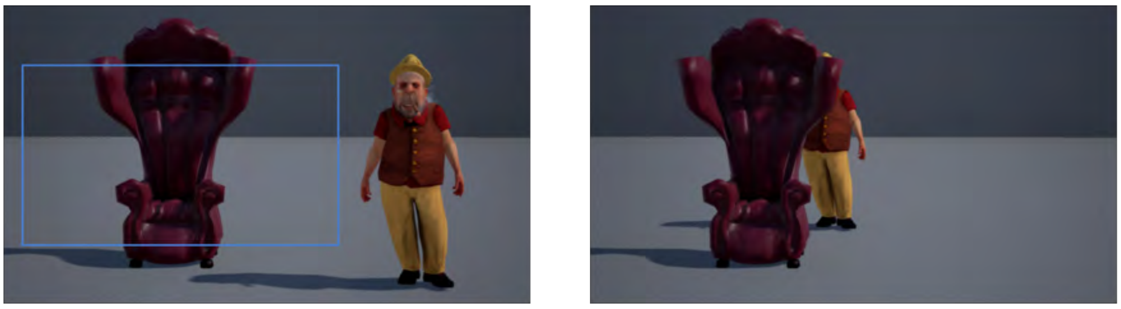


Figure 7.4: Illustration of the two types of visibility consideration. In the left image, if the frame is cropped to be within the bounds of the blue rectangle, the character will be beyond the frame bounds and therefore not visible. In the right image, the camera's view of the character is mostly occluded by the chair and the character is therefore not visible. The character is only visible if they are within the bounds of the frame and not occluded.

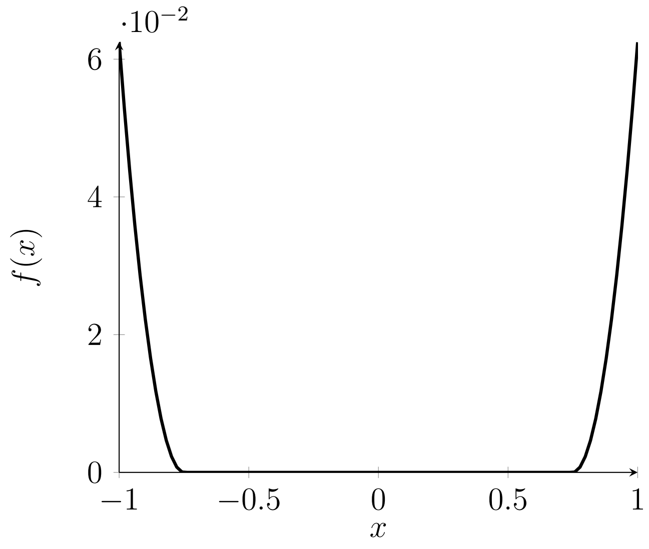


Figure 7.5: Visibility function plot.

where

$$g(x) = \begin{cases} (x - x_{\text{low}})^2 & x < x_{\text{low}}; \\ 0 & x_{\text{low}} \leq x \leq x_{\text{high}}; \\ (x - x_{\text{high}})^2 & x > x_{\text{high}}. \end{cases} \tag{7.2}$$

7.2.2.2 Occlusion and Collision

The occlusion rule is simply that there must be an unoccluded line of sight from the camera to an object that the user desires to be visible. A common distinction is made between clean shots where all of an object is unoccluded and dirty shots where only part of an object is unoccluded.

Similarly, the collision rule simply states that the camera must not collide with scene elements. Just as it is generally undesirable for real world cameras to pass through walls and actors, users of virtual cameras generally expect their cameras not to pass through apparently solid virtual matter.

Unfortunately, building suitable objective functions for occlusion and collision is much more complicated. Unlike all of the other functions described, these objectives represent an observation about the geometry of the scene as a whole and therefore cannot be defined in the same concise analytic style used elsewhere. Furthermore, there will be a discontinuity in any direct observation of visibility at the boundary of occlusion and collision. To address these issues, the system provides an n-dimensional linear interpolator that, given camera position $\mathbf{c}$ and point position $\mathbf{p}$, can be expressed as

$$f(\mathbf{c}) = g_3(\mathbf{c}), \tag{7.3}$$

where

$$g_i(\mathbf{c}) = \frac{\mathbf{c}_i^a - \mathbf{c}_i}{\mathbf{c}_i^b - \mathbf{c}_i^a} g_{i-1}(\mathbf{c}^b) + \frac{1 - \mathbf{c}_i^a + \mathbf{c}_i}{\mathbf{c}_i^b - \mathbf{c}_i^a} g_{i-1}(\mathbf{c}^a). \tag{7.4}$$

Here, $\mathbf{c}^a$ and $\mathbf{c}^b$ are the closest sample locations on axis i such that $\mathbf{c}_i^a \leq \mathbf{c}_i$ and $\mathbf{c}_i^b \geq \mathbf{c}_i$ and $\mathbf{c}_j^a = \mathbf{c}_j^b = \mathbf{c}_j, \forall j \neq i$, and

$$g_0(\mathbf{c}) = B(\mathbf{c}, \mathbf{p}), \tag{7.5}$$

where

$$B(\mathbf{c}, \mathbf{p}) = \begin{cases} 1 & \text{if the path from } \mathbf{c} \text{ to } \mathbf{p} \text{ is occluded;} \\ 0 & \text{if the path from } \mathbf{c} \text{ to } \mathbf{p} \text{ is not occluded.} \end{cases} \tag{7.6}$$

As this function is not analytically differentiable, differentiation is computed numerically as $\frac{df(\mathbf{x})}{dx} \approx \frac{f(\mathbf{x}+\boldsymbol{\Delta}_x)-f(\mathbf{x})}{\boldsymbol{\Delta}_x}$ with $\Delta_x = 1\,\mathrm{cm}$.

To improve smoothness, the system supports the convolution of the samples of the interpolator by a Gaussian kernel, which can be expressed mathematically as changing $g_0(\mathbf{c})$ to

$$g_0(\mathbf{c}) = \frac{1}{\sqrt{2\pi\sigma^2}} h_3(\mathbf{c}). \tag{7.7}$$

Here,

$$h_i(\mathbf{c}) = \sum_{j=-K/2}^{K/2} e^{-\frac{(j\Delta_i)^2}{2\sigma^2} h_{i-1}(\mathbf{c}+j\Delta_i U_i)}, \tag{7.8}$$

where Δ is the set of convolution step sizes, U is the set of bases for $\mathbf{c}$, and $h_0(\mathbf{c}) = B(\mathbf{c}, \mathbf{p})$. Values $K = 7$, $\sigma = 1$, and $\Delta = 25\,\mathrm{cm}$ are used. Of course, the convolution requires more samples than its unconvolved counterpart, decreasing efficiency.

Note that in all uses, $B(\mathbf{c}, \mathbf{p})$ is sampled in a uniform grid with steps that are specified at function construction. Samples in the current search neighborhood can be fetched on demand, then cached in a hashmap for efficient retrieval in later computations. This approach differs from (Drucker and Zeltzer, 1995) in that it requires no preprocessing of the scene geometry, does not require the occluding or colliding geometry to remain static, and can scale to scenes of any size, given a utility for sampling $B(\mathbf{c}, \mathbf{p})$ that is equally efficient in scenes of any size.

7.2.3 Shot Size

Modeling shot size is a bit tricky. While there are several different ways of evaluating this rule using projective geometry, most tend to suffer from severe instability given even moderately suboptimal inputs. For example, the function $f(\mathbf{a}, \mathbf{b}) = (1 - |\Pi(\mathbf{a})_y - \Pi(\mathbf{b})_y|)^2$, where the subscript denotes the y component of the vector, is highly unstable.

Given object points $\mathbf{a}$ and $\mathbf{b}$, camera position $\mathbf{c}$, and angle θ, the most stable objective function we have found is

$$f(\mathbf{a}, \mathbf{b}, \mathbf{c}, \theta) = \left(\theta - \arccos\left(\frac{(\mathbf{a} - \mathbf{c}) \cdot (\mathbf{b} - \mathbf{c})}{||\mathbf{a} - \mathbf{c}||\; ||\mathbf{b} - \mathbf{c}||}\right)\right)^2. \tag{7.9}$$

As described above, the object points **a** and **b** are chosen to be the edges of the object to be in frame (e.g., top of head and mid-chest for a close-up shot). For most purposes, the value of θ can be chosen to be the camera's vertical field of view. However, when headroom is desirable for the same points (see Section 7.2.2.1), this must be decreased slightly so as to avoid unnecessary objective conflict.

7.2.4 Relative Angles

Modeling relative angles is relatively simple using spherical geometry. Given object point **p**, camera position **c**, unit object up direction **u**, and desired angle θ relative to unit object look direction **d**, the objective function for vertical angle is

$$f(\mathbf{p}, \mathbf{c}, \mathbf{u}, \theta) = \left(\theta - \arccos\left(\frac{\mathbf{p} - \mathbf{c}}{||\mathbf{p} - \mathbf{c}||} \cdot \mathbf{u}\right)\right)^2, \tag{7.10}$$

while the profile angle function is

$$f(\mathbf{p}, \mathbf{c}, \mathbf{u}, \mathbf{d}, \theta) = \left(\theta - \arccos\left(\frac{\mathbf{p} - \mathbf{u}((\mathbf{p} - \mathbf{c}) \cdot \mathbf{u}) - \mathbf{c}}{||\mathbf{p} - \mathbf{u}((\mathbf{p} - \mathbf{c}) \cdot \mathbf{u}) - \mathbf{c}||} \cdot \mathbf{d}\right)\right)^2. \tag{7.11}$$

7.2.5 Rule of Thirds

Attempting to formulate a function that intuitively models the rule of thirds is a bit tricky. Clearly, a function $f : \mathbb{R} \leftarrow \mathbb{R}$ with minimums at $\pm 1/3$ is desired, but many such functions exist.

One simple formulation can be expressed as $g(x) = \frac{x^4}{x_0^4} - \frac{2x^2}{x_0^2} + 1$. This has the desired $\pm 1/3$ minimums with $x_0 = 1/3$, but penalizes inputs for which $|x| \leq x_0$ much more lightly than those for which $|x| > x_0$. Because of the exponential nature of this asymmetric penalization, this formulation tends to be tricky to balance with other desirable properties.

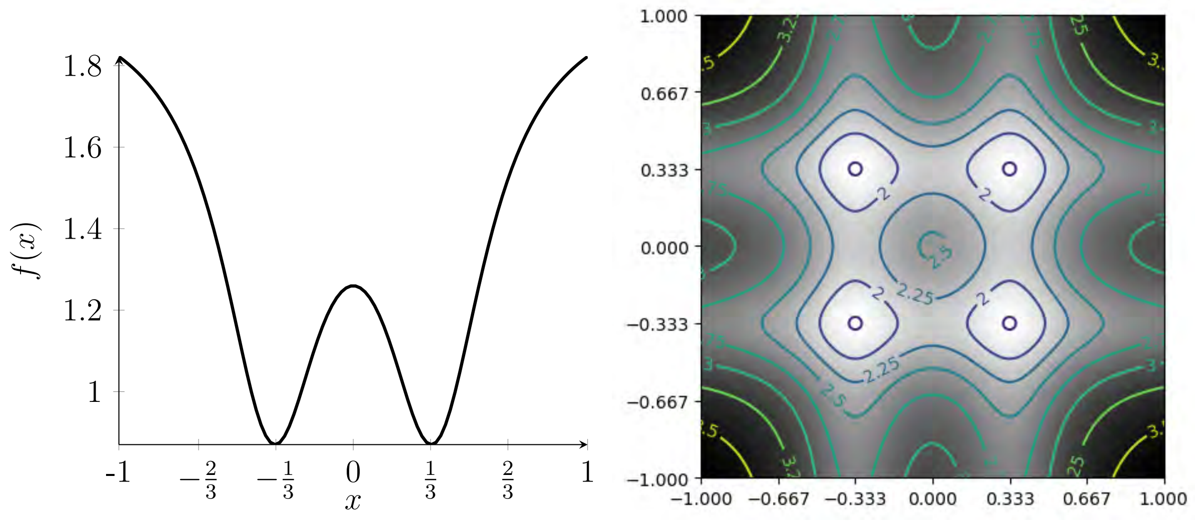


Figure 7.6: A 1D (left) and 2D (right) plot of the rule of thirds function.

To combat this, we defined a "flatter" formulation. Given a point $\mathbf{p}$ in world space as well as constants $0 \leq x_0 \leq 1$ and $0 < a \leq 1$, this objective is modeled as

$$f(\mathbf{p}) = \sum_{x \in \Pi(\mathbf{p})} \frac{(x+b)^2}{(x+b)^2 + a} + \frac{(x-b)^2}{(x-b)^2 + a} \tag{7.12}$$

where

$$b = \sqrt{\frac{2\left(x_0^2 - a\right) + \sqrt{4\left(x_0^2 - a\right)^2 + 12\left(x_0^2 + a\right)^2}}{6}}. \tag{7.13}$$

To use the standard third lines, x_0 should be set to 1/3. Selecting a value for a is a bit trickier, as too high a setting causes the function to affect only points in the immediate vicinity on-screen, while too low a setting can cause the penalty for $|x| < x_0$ to decrease significantly. It was found experimentally that $a = 0.07$ produced satisfactory results.

This formulation has an issue in that it penalizes points far from x_0 equally as $\lim_{x \to \pm\infty} g(x) = 2 - h(x_0)$. In some applications, this "flatness" is desirable; for example, if there is an object that is not required in a shot, but that should be placed on the third lines when it is in frame.

7.2.6 Look Space

Given object point $\mathbf{p}$, unit object look direction $\mathbf{d}$, camera position $\mathbf{c}$, and unit camera right direction $\mathbf{c}_R$, the look space objective can be evaluated using the visibility frame bounds objective function $f_{\text{vis}}(\mathbf{p})$ from Section 7.2.2.1, as

$$f(\mathbf{p}) = f_{\text{vis}}\left(\Pi(\mathbf{p}) + \frac{\Pi(\mathbf{p}+\mathbf{d}) - \Pi(\mathbf{p})}{||\Pi(\mathbf{p}+\mathbf{c}_R) - \Pi(\mathbf{p})||}\right). \tag{7.14}$$

7.2.7 180°rule

The 180°rule, sometimes called the line or plane of axis rule, can be modeled as a simple geometric function. Given camera position $\mathbf{c}$, unit up direction $\mathbf{u}$, and two points $\mathbf{a}, \mathbf{b}$, the objective function

$$f(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{u}) = \max(0, ((\mathbf{b}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a})) \cdot \mathbf{u})^2. \tag{7.15}$$

Assuming we are operating in a left handed coordinate space, this will specify that the camera should be on the left side of the plane of action, although swapping the order of $\mathbf{a}$ and $\mathbf{b}$ allows either side of the plane to be specified.

7.2.8 Depth of Field

Focal length f, as measured in the same units as world space. Lens f-number n, which is the ratio of the focal length to the entrance pupil diameter. Focus distance s. Given all of this, the depth of field can be modeled by calculating the circle of confusion for camera space point $\mathbf{p}$ as

$$c(\mathbf{p}, s, f, n) = \frac{|||p|| - s|}{s}\frac{f^2}{n(s-f)}. \tag{7.16}$$

A satisfactory objective function for depth of field can then be described with the same visibility frame bounds objective function $f_{\text{vis}}(\mathbf{p})$ from Section 7.2.2.1 as $f(\mathbf{p} = f_{\text{vis}}(c(\mathbf{p}) - m)$, where m is the desired circle of confusion size.

Type	Eqn.	Weight	Input Parameters
Rule of Thirds	(7.12)	0.5	Image space character head position
Relative Angle - Vertical	(7.10)	200	Head position, Global up direction, Desired angle
Relative Angle - Profile	(7.11)	300	Head Location, Head Forward, Desired Angle
Shot Size	(7.9)	750	Top and Bottom Character positions, Camera FOV offset
Visibility - Frame Bounds	(7.1)	10	Image space character top and bottom positions, Frame bounds offset (e.g., 0.1)
Visibility - Occlusion	(7.3)	700	Character position, σ (defaults to 1)
Look Space	(7.14)	5	Character head position, Character head forward direction, Frame bounds offset (e.g., 0.3)

Table 7.1: Recipe for Objective Function Corresponding to a Cinematographic Composition With a Single Person

7.3 Basic Cinematographic Recipes

If the objective functions described in the previous section are ingredients, what kinds of recipes can be used to prepare an objective matching a combination of desired cinematic properties?

One of the most stable and most versatile cinematographic recipe I have found corresponds to any composition containing a single character viewed from a particular angle, represented by a simple weighted sum of the following objectives, and is summarized in Table 7.1. Shot size can be controlled by specifying what points on the character should be near the top and bottom of the frame.

Note that in any weighted sum, the scale of the weights is less important than the ratios between pairs of weights. If, hypothetically, all the weights were to be scaled by a factor of 100, the relative optimality of inputs would remain invariant, although the rate of convergence and numerical stability of different optimization algorithms might change. However, modifying the ratio between weights can dramatically affect what inputs are locally or globally optimal, thereby affecting the relative importance of different

Type	Eqn.	Weight	Input Parameters
Relative Angle - Vertical	(7.10)	200	Position of midpoint between character heads, Global up direction, Desired angle
Relative Angle - Profile	(7.11)	200	Position of midpoint between characters, Direction of axis between characters, Desired Angle
Shot Size	(7.9)	750	Positions of both characters' heads, Camera FOV offset
Visibility - Frame Bounds	(7.1)	10	Image space top and bottom positions for both characters' heads, Frame bounds offset (e.g., 0.05)
Visibility - Occlusion	(7.3)	700	Both character head positions, σ (defaults to 1)

Table 7.2: Recipe for Objective Function Corresponding to a Two-Person Cinematographic Composition with Controls Analogous to the Toric Space

cinematographic properties in the final composition. The weights listed in the table above seem to be both numerically stable[123] and produce a relatively smooth objective function.

One notable pattern in the weights listed above is that objectives that compute values in image space tend to require much lower weights than those operating purely in world space. This is a consequence of the increased sensitivity image space objectives place on rotational parameters over positional parameters. While the effect of camera translation on the image space position of an object (e.g., rule of thirds, frame bounds visibility, look space) varies with the object's distance, rotating the camera even a small amount will significantly shift the image space position of an object at any distance.

As another example, Table 7.2 contains a recipe I have found to be roughly analogous to the toric space controls, which is specifically useful for scenes with two on screen actors, developed by Galvane et al. (2015a); Lino and Christie (2015).[124] In this recipe, the same objectives for shot size and relative angles can be easily adapted to map from a single actor to modeling compositional properties between a pair of actors.

[123]Implemented with double precision floating point numbers.

[124]See sections 3.1.1.1 and 3.1.2.2 for further discussions of toric space.

One of the exceptionally nice features of this recipe style is its modularity. If a user does not like having some predefined objective, they can replace it with another objective, such as one of the frame position functions, or remove it entirely. If they feel something is missing, like depth of field or the 180°objectives, they can simply and easily add to the recipe.

CHAPTER 8

Machine Learning Defined Cinematographic Objective Functions

With so many films being made every year, why do we need to adapt and invent such sophisticated and delicate mathematical machinery to capture desirable cinematographic properties? Is it possible to construct a camera control system that learns how to make cinematographic decisions by merely watching existing movies? This dream of data-driven virtual cinematography is incredibly enticing, but far from easy.

As discussed in chapters 3, 2, and 6, motion pictures represent a challenging medium. Each moment of the film is the result of a multitude of creative and technical decisions, from what events are portrayed to how the actors play their roles, as well as how to move the camera to best capture the emotional and informative intent of the scene, among many others. However, rather than being directly presented with details on the motivation and decision making process for each of these tasks, what is presented to us are sequences of images, with varying numbers of people in diverse compositions to communicate changing messages over time.

As a result, it can be unclear how to interpret this unannotated data to control a camera in an existing scene. For example, consider the case of a user wanting to copy a composition from a film clip with multiple moving actors to a target camera in another virtual scene. If the actors' motions in the clip precisely match the motions in the target scene, we can consider directly transferring the camera motion. However, if there is any significant motion mismatch, perhaps the actors have different sizes or are facing different directions, the virtual cinematography system will be forced to automatically compromise between competing observed camera behaviors. Are the positions of the actors in the frame more or less important than their relative angles? Are look space or head room more important for one actor than another? How important are the sizes of the actors relative to the other properties? Worse still, these and similar questions grow even more complicated when considering a model capable of reasoning about a variable number of subjects within the composition.

Faced with such challenges, how can a general purpose strategy to learning virtual cinematography from data be formed?

8.1 Data Representation

As explored in the Section 6, I now have at my disposal a large and novel dataset concerning cinematographic composition with human subjects. However, there are several problems in data representation, as the dataset is not inherently uniform.

1. The frames have varying resolutions, aspect ratios, and with differing dimensions of values for different parameters collected. This can be easily addressed by using normalized display coordinate space, and normalizing all angles into a $[0, 1]$ or $[-1, +1]$ range.

2. All of the machine learning models I have experimented with require fixed dimension inputs with normalized ranges, but the dataset consists of a variable number of parameters collected from each frame. This parameter count variability is dependent on two factors: what computer vision tools triggered for each detected person, and the number of people detected.

   (a) To address the problem of differing detectors firing for different people in different frames, I experimented with three different approaches: fill missing values for desired parameters with zeros, only use parameters that are mutually available, and try to learn a model capable of filling in missing parameters. I have so far been unable to identify a significant advantage for any of these options in my experiments, so have defaulted to filling in missing values with zeros as the easiest option.

(b) The problem of varying numbers of detected people in each frame can be solved by specifying an input space with channels for N distinct detected people. In this formulation, each person channel would have the same input feature dimensions, which are set to zero if no person is occupying this channel in this frame.

This leaves a problem of how to order detected people from the frame into the available channels. The solution used here, which was pioneered by (Jiang et al., 2020), sorts people by their average size over all the frames in the shot. This is based upon a cinematographic rule of thumb commonly credited to Alfred Hitchcock, which asserts that the largest person in frame is usually the most important. This Hitchcock rule ordering not only keeps each person in the same channel throughout multiple shots in the same frame, but also ensures that the largest people appear in the first slots most often. This is why having a consistent measure of person size was such an important part of the data collection process.

8.2 Modeling strategy and motivation

With these techniques, the dataset can be formed into a fixed dimension format with normalized inputs. However, forming a coherent machine learning model around this data is a tremendously unintuitive task, as what we essentially have is an unsupervised learning task.

Unlike classic machine learning tasks like classification or regression, we have no obvious cinematographic properties to attempt to learn here. Worse still, it is unclear whether attempting to classify compositions into individual cinematographic categories, such as shot size or emotional intent, would be useful for powering complex camera control behavior, as doing so would not inherently inform any balance or compromise mechanism between a user's desires among multiple categories.

Also, the goal of virtual cinematography used here is to control a virtual camera in an existing 3D space using the positions and orientations of objects in the scene. As discussed in Section 3.2, there are a variety of popular and distinct problems that do not meet this definition, such as image synthesis and pixel-based aesthetics analysis, have widely utilized machine learning techniques.[125]

Alternatively, it can be tempting to imagine, given the end goal of crafting a practical camera controller, that the ideal machine learning model is inherently generative: pick some camera control space (e.g., Cartesian, polar, or toric space), and try to learn a mapping directly from actor motion to camera pose/path in that camera control space. While this is an intuitive approach that can work in some cases, it is far from ideal. This is primarily because the data available does not contain camera motion in anything like a global reference frame, instead only capturing the motion of actors relative to the camera. Some other researchers have addressed this limitation by choosing a camera control space that is defined relative to the actor(s) in the scene. However, such a controller could then only be used when the same type and number of actors appear in the scene, severely limiting the usability of the resulting camera system.

Instead, the approach I have taken is to consider a model that has learned a distance metric between some user specified cinematographic intent and a prospective composition. Were we to have such a model, all we would have to do is ask the optimization tools propounded upon at length in Section 5 to find the optimal composition with the lowest distance to the desired composition. Not only would such a model be compatible with all of the rest of the mathematical tools outlined so far, including smoothing, the model itself could provide valuable analytical insight into the grammar or semantics of the language of cinematographic composition.

It should be noted that I have separated the tools used for machine learning from

[125]While the generation or manipulation of pixels is of a fundamentally distinct nature to the problems explored here, future research into techniques such as inverse rendering may allow these problems to be bridged. See Section 3.2 for a longer discussion.

those for camera control. Except where otherwise noted, I have used PyTorch ((Paszke et al., 2019)) for the training of machine learning models. These trained models are then exported into a JSON file format, which can be parsed and imported into my optimization framework for camera control. This separation allows for PyTorch GPU acceleration when no real time graphics integration is required, and the existing integrations between my optimization system and the real time graphics system to be used when needed.

8.3 PCA

The first option I explored was simple principal component analysis (PCA).

Taking a set of 5 parameters including head position (i.e., NDCS X and Y), head height, and head orientation (i.e., pitch and yaw), for each detected person in isolation, PCA reports that all 5 dimensions are needed to account for at least 95% of the variance in the dataset. Interestingly, adding in shoulder and pelvis orientation to a PCA reports that only 7 dimensions are required, despite this adding an additional 4 inputs. PCA seems to indicate that vertical and horizontal position are the two most significant dimensions from the original dataset.

Taking multiple people, as intended by the Hitchcock rule ordering, we can run PCA on compositions with multiple people. Again, this analysis was done with the same 9 input dimensions as used in the previous analysis, but also with an additional input for each person indicating whether this person was detected at all. Additionally, all missing values were filled with zero.

Intuitively, increasing the number of people in the composition requires increasing numbers of PCA dimensions. However, these increases are not uniform. While each of the first two people in the composition seem to require 7 additional dimensions, each subsequent person seems to require only 5 or 4 additional dimensions. I suspect this is primarily because most compositions in the dataset contain two or fewer people, meaning that less variance is captured by each person past the second.

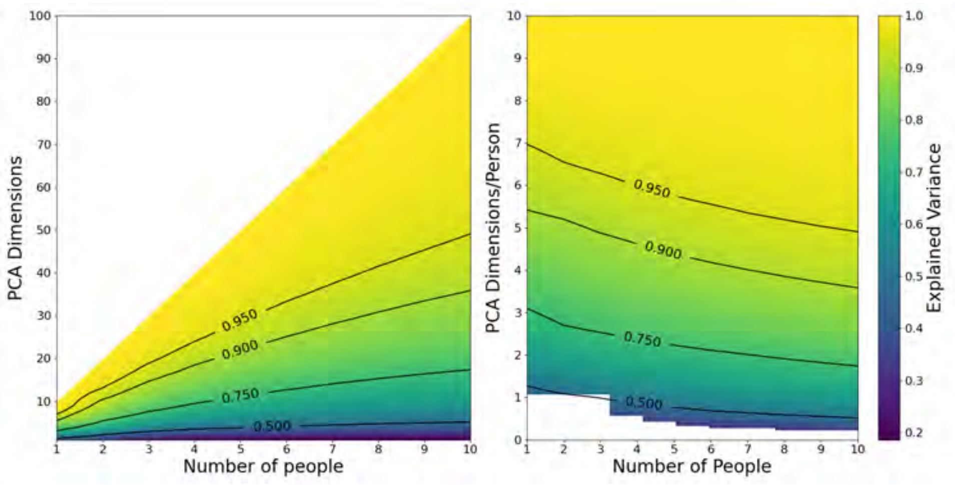


Figure 8.1: Plot of variance explained by PCA for collected data.

I also dabbled with clustering this data, both before and after PCA dimensionality reduction. Unfortunately, the dataset is simply too large for me to do any kind of thorough exploration of the clustering. Running K-means with varying parameters only really revealed that compositions could be clustered by the number of people detected within them, which is far from insightful.

While much of this is analytically interesting, it is not remotely clear how to leverage this information in the manufacturing of a practical camera controller.

8.4 Frame Embedding

So far, we have largely ignored the only obvious structure in this data: the sequential nature of frames. Given that our dataset consists of data captured from frames sampled at 0.5 second intervals from the original video, frames in the same shot that appear close together in time are likely to correspond to similar compositions. Conversely, if two frames from the dataset are chosen at random, we would expect dissimilar compositions with high probability.

Essentially, this forms the basis for a *frame embedding* task, where the goal is to find a mapping from data space to some latent space in which pairs of frames believed to be similar have a small distance, while those believed to be dissimilar have a large distance.

This concept is inspired by word embedding, a common technique in natural language processing, which represents a similar task with textual words in a plaintext corpus. However, there is a key difference between our frame embedding task and word embedding: words are discrete and finite, while our input vectors are essentially continuous. Whereas many standard word embedding techniques rely upon counting the relative coincidence of words, the continuous vectors from our dataset cannot be made discrete in any obvious way.

Instead, we can provide a more literal interpretation of the motivation. Suppose we have two frame representation vectors $\mathbf{x}$ and $\mathbf{y}$, and $f : \mathbb{R}^a \to \mathbb{R}^b$ that maps frame representation vectors into a latent space. We would like to be able to say that the distance between the latent space vectors should be as small as possible if the compositions these vectors represent are similar, while we would like the distance between these vectors to be at least some preset value if they are dissimilar.

I chose to model this as a loss function

$$L(\mathbf{x}, \mathbf{y}, a, b) = ELU\left(a\left(|f(\mathbf{x}) - f(\mathbf{y})| - b\right)\right) + 1$$

where the exponential linear unit function

$$ELU(x) = \min(|x|, e^x - 1).$$

In this formulation, weight label a is $+1$ if the two embedding space vectors are similar and -1 if the vectors are dissimilar, while weight label b corresponds to the desired minimum distance between dissimilar vectors.

With this **minimum distance loss (MDL)**, I chose a simple network architecture with 4 linear layers sandwiching leaky-ReLU layers and leading to a 10-dimensional embedding space, and proceeded to try to train the model. The dataset was sampled such that frames within a shot in positions ± 3 of a given frame were considered similar, while 25 other frames from the dataset were chosen to represent dissimilar frames, and assigned

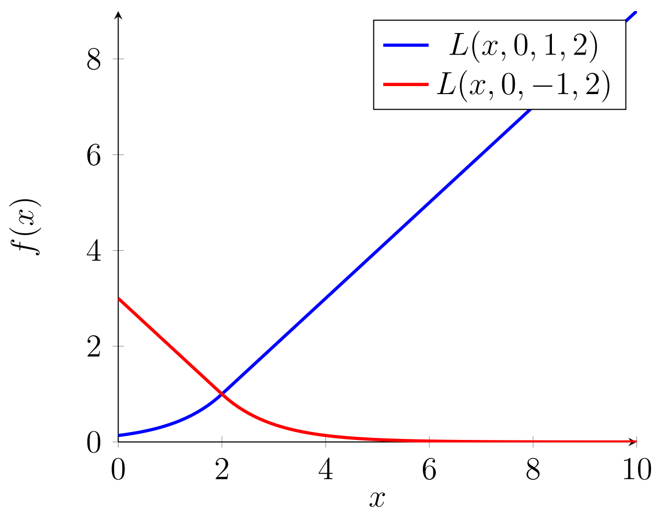


Figure 8.2: A Plot of a 1D version of the MDL loss function.

weight values of $a = -1, b = 3$. After removing pairs of frames where either frame was empty (i.e., contained no detected people), this resulted in a training dataset with ~ 18 million pairs of frames, where $\sim 10.6\%$ were similar, and the remainder were dissimilar.

After training a model in this way, an early issue appeared in the form of a profoundly disadvantageous interaction between sampling distributions and input parameters chosen. As previously mentioned, the films all have varying fundamental characteristics that I intentionally attempted to normalize away, one of which was aspect ratio. Although it did not appear anywhere in the input data I used to train, when I started plotting the results of my first experiments, what I discovered was that aspect ratio seemed to be one of the principal input parameters the model was using in its mapping.

This partially makes sense: as a considerable portion of the dissimilar frames were sampled from films with different aspect ratios, the model would have a large incentive to use the aspect ratio as a key parameter. However, neither the aspect ratio nor resolution of the frame were parameters given to the model, begging the question of how the model seemed to be using this data. However, the initial input vectors contained both the height and the width of detected heads, which tend to be almost perfectly linearly correlated in pixel space. When the data was normalized to normalized display coordinate space, the ratio between these values becomes indicative of the frame aspect ratio, allowing the model to learn an intentionally omitted property.

As a result of this behavior, this model did not seem to be forming a mapping that unified data from all the films, instead partitioning films with differing aspect ratios into separate regions of the latent space. I made two changes to combat this problem. First, I removed the head width from the input vector, leaving only height. Second, I added 9 randomly chosen frames from neighboring shots in the same film to the set of dissimilar frames for a given frame, with weight values $a = -1, b = 2$. After again removing empty pairs of frames, this resulted in a training dataset with $\sim$ 21 million pairs of frames, where $\sim$ 9.2% were similar, and the remainder were dissimilar.

These changes produced a dramatic improvement, albeit one that is hard to quantify. In this new model, not only does aspect ratio no longer seem to be strongly correlated with distance in the embedded space, but many of the resulting feature distances seem to make sense.

The resulting mapping is fascinating to manually explore, and several views of a PCA projection from this embedding space to 3D are picture in Figure 8.3. Just from the views shown, it is clear that embedding distance is highly correlated with the number of people detected in the frame, and moderately correlated with the position of the most significant person in the frame. However, the relative angles of the most significant detected person show only minimal evidence of being correlated with embedding distance, suggesting that the property often matters little for single person compositions.

However, this mapping only utilizes a small part of the sequential information provided. As explored in Section 2.4, a significant portion of film semantics is captured in the order and timing of sequences of shots edited together.

I tried to capture some of the sequential shot information missing in the MDL model with another frame embedding model inspired by the Word2Vec model now ubiquitous in NLP (Mikolov et al., 2013). This **Frame2Vec** paradigm models the distance between frames as a cosine loss between the latent embedding space for a given shot, and a latent context space for any shots in neighboring shots. Formally, this loss function was

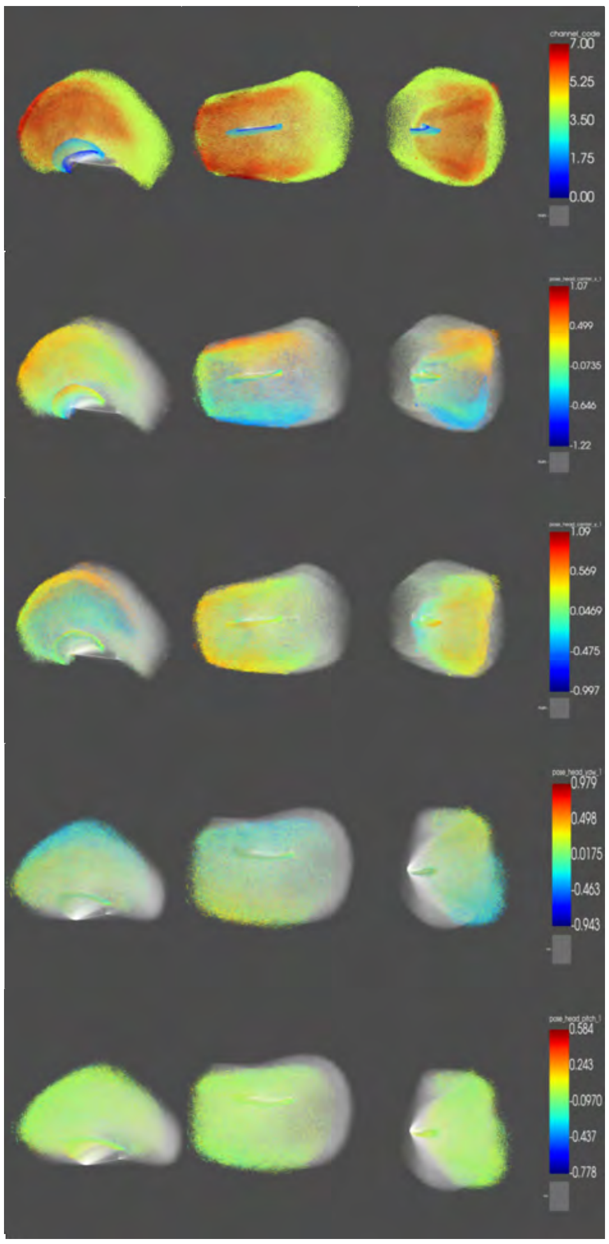


Figure 8.3: Visualization of the MDL frame embedding. These plots visualize the result of using an MDL model to transform the $\sim$ 1M frames from the dataset into the latent embedding space of the model, in which the distance metric can be applied. The 10 dimensional latent space was then reduced to the most significant 3 dimensions with PCA for visualization. Here, coloring by 5 properties are rendered from 3 different axis-aligned views, with each row corresponding to a single property and each column corresponding to a single view. The first row is colored according to what person channels are used in each frame, a property referred to as channel code. The second and third rows are colored according to the x and y components of the image position of the head of the most significant detected person. The last two rows are colored according to the yaw and pitch of the most significant detected person.

represented as

$$L(\mathbf{x}, \mathbf{y}) = BCE(\sigma(f(\mathbf{x}) \cdot f(\mathbf{y})), \ell)$$

where the sigmoid function

$$\sigma(x) = \frac{1}{1 + e^x}$$

and the binary cross entropy function

$$BCE(x, \ell) = -(\ell \log(x) + (1 - \ell)\log(1 - x)).$$

Using a similar distribution of pairs of frames from the MDL set, but with labels of 0 or 1 as targets for cosine loss. Unfortunately, I have yet to discover any additional capabilities this Frame2Vec model provides over the MDL model described earlier, although the mapping it produces is distinct.

8.5 Limitations

However, both the MDL and Frame2Vec embedding models suffer from a profound lack of expressivity when implemented as objective functions for camera control optimization problems. Any user specified composition has only one representation in each model, which means that a user cannot modify their preferences regarding the relative importance of any part of the composition without modifying the composition itself. To put it another way, this model allows a user to specify where an optimum will be in the resulting objective functions, but does not allow the user to modify any of the rest of the function, including compromise behavior.

For example, when I first incorporated the resulting MDL model into a camera controller, I discovered that the model did not prioritize the relative angle of the head nearly as much as I would personally prefer. For some scene geometries, the resulting camera did not seem to want to care about relative angles at all, instead choosing whatever orientation would put the actor into the desired position and height in the resulting frame.

One interpretation of this problem is that this frame embedding paradigm fails to take into account the fact that, although each frame might have some semantic or aesthetic meaning individually, much of the intent behind cinematographic decision making is hidden in the temporal grouping of frames. From a single frame in a shot, no observer would be able to accurately determine what the distribution of properties the rest of the frames in the same shot might have. As each frame has only one frame embedding regardless of the temporal context in which the frame appears, any temporal relationships seem to be uncapturable in this frame embedding paradigm.

While I have done some preliminary investigations into additional models with the potential to overcome these issues, none have so far produced results of sufficient significance for deep discussion in this text. The following includes a few of these exploratory ideas.

- Bag-of-frames: Instead of using only a single frame for context, use a sequence or set of frames surrounding the current target frame to capture the temporal semantics currently utilized. While appealing, this presents several issues, such as how to structure the model so that differing numbers of frames can be input, and what training distribution will allow for the most structure to be learned without introducing undesirable bias.

- Transformer: A variety of recurrent machine learning models, such as transformers, have been shown to be highly effective at capturing sequential semantics in natural language tasks. While such a recurrent model could be used as the basis for a declarative virtual cinematography distance metric, doing so would likely limit the flexibility of the other techniques utilized here, as the change from instantaneous to sequential temporality would make optimization order with varying keyframe density much trickier.

- Weight learning: Instead of learning a mapping from frame representations to latent space, another alternative would be to learn a mapping to weights for predefined objective functions from Chapter 7. This has the appealing advantage that it could improve controllability, as the lack of explainability of the preceding black-box systems are likely to make controllability a challenge. However, how to structure these functions for compositions with a varying number of people remains unclear.

In addition to these possibilities for future machine learning research, another exciting opportunity is present in using the data to determine characteristics common to the cinematography of different categories of films, such as what distinguishes the styles of different directors or genres. While some researchers, such as Courant et al. (2021), have performed limited explorations of this topic for pixel-based cinematography, no one has yet explored these questions with purely vector-based data.

CHAPTER 9

Results

All of the described tools and techniques were implemented in a single-threaded prototype camera control system that was integrated with *Unreal Engine 4.25*. This system can either be run online with the engine's standard game thread, or in a parallel thread that shares results as the optimization algorithm progresses with the main game thread. Several different experimental scenes, with several different objective functions and optimization strategies, were then tested for both scripted and unscripted applications.

In most experiments, each camera pose required that the system optimize over 5 variables: position in $\mathbb{R}^3$, yaw, and pitch. All experiments had a static user-specified aspect ratio, specifically a ratio of ~ 1.78 width to height commonly referred to as 16:9. A few experiments also included field of view, focus distance, and aperture as variables to optimize over, but most experiments used user-specified values. In principle, the system is capable of optimizing over both aspect ratio as well as camera roll, but no experiments run have yet called for either.

After some empirical experimentation, different combinations of desired compositions and scenes seem to call for different keyframe rates. Generally speaking, wider compositions of slower scenes require fewer keyframes than tighter compositions of faster scenes with more abrupt motions, with typical keyframe rate values falling between 2 and 40 per second. However, one of the wonderfully elegant features of instantaneous objective functions is that the keyframe density of the resulting optimization problem can be nonuniform and dynamic, with the same optimization setup suited relatively equally well to a range of densities.

9.1 Scenes

While most of the scenes used in these experiments take place in simple environments, the motions of the actors within these scenes comprise a variety of actions. These actor motions have two sources.

Some of the scenes, especially those where the actors are interacting closely with the environment, are simply the result of automatically animated game characters reacting to some combination of player input and preprogrammed game AI. As these **game character** motions come directly from the game engine, they are trivial to integrate with the game engine.

The remainder of the scenes pull actor motion from publicly available motion capture datasets, in particular the CMU mocap (Cmu, 2022) and Human3.6M (Ionescu et al., 2014, 2011) datasets. Whereas many of the game character motions tend to be cartoonishly exaggerated, these **mocap** motions offer the realistic and subtle expressions of human performers.

However, I found the original mocap data sources to be quite noisy, producing unnaturally jittery motions when applied to game character meshes. To mitigate this issue, I applied a smoothing operation to the joint rotation data, setting the orientation for a joint at a keyframe to be equal to the average of that joint's orientation within ± 3 keyframes.

Correctly computing the average of orientations is a highly nontrivial problem. To circumvent some of this difficulty, I used a modified version of an existing python library ((Hagen, 2017)), which in turn was an implementation of (Markley et al., 2007). The approach taken models orientations as quaternions, $\mathbf{q}_1, \dots, \mathbf{q}_n$, where each quaternion $\mathbf{q}_i$ contributes to the average with weight w_i such that $\sum_{i=1}^{n} w_i = 1$. In this approach, the weighted average can be computed as the eigenvector with the maximum eigenvalue of the matrix

$$A = \sum_{i=1}^{n} w_i \mathbf{q}_i \otimes \mathbf{q}_i,$$

where $\mathbf{q} \otimes \mathbf{q}$ is the outer product of $\mathbf{q}$, treated as a real vector, with itself. This process is somewhat computationally expensive, so it was desirable to run it once, offline from any experiments, and save the smoothed animation data to disk.

Finally, these smoothed mocap motions must be retargeted from their original motion capture performer's skeletons on to whatever skeletal mesh is desired within the game engine. My methodology for this was relatively simple, with the simple goal of guarantee rotational alignment between certain important character bones in the unmodified reference poses for the original and destination skeleton. For example, most of the target skeletons from the mocap datasets had a pure T-pose as reference, while most of the game engine skeletons use a pose with arms and legs at a more natural angle. Aligning these bones can be accomplished with a single depth first pass through each skeleton, computing the relative local quaternion rotation required to align each important bone as each joint tree is descended. These local retargeting transformations can then be applied to each source joint orientation when loading animation data for this skeleton.

Separately, I also briefly experimented with trying to use the motion of actors from my own dataset, but found the results less than satisfactory. While the dataset contains 3d joint position estimates, all motions are relative to the camera, which could be moving, and only captured at 2FPS. Personally, I found the results of animating a 3D character with these values to be uncanny at best and unrecognizable at worst.

9.2 Use Cases

To demonstrate the effectiveness of the system across different use cases, camera paths from two sample scenes are visualized in Figure 9.1. These use cases are divided into the following categories.

1. Unscripted: The system computes a camera pose for each frame online with the display of the frame, and with zero future knowledge. The objective is optimized with a gradient based algorithm (e.g., gradient descent, backtracking line search) for

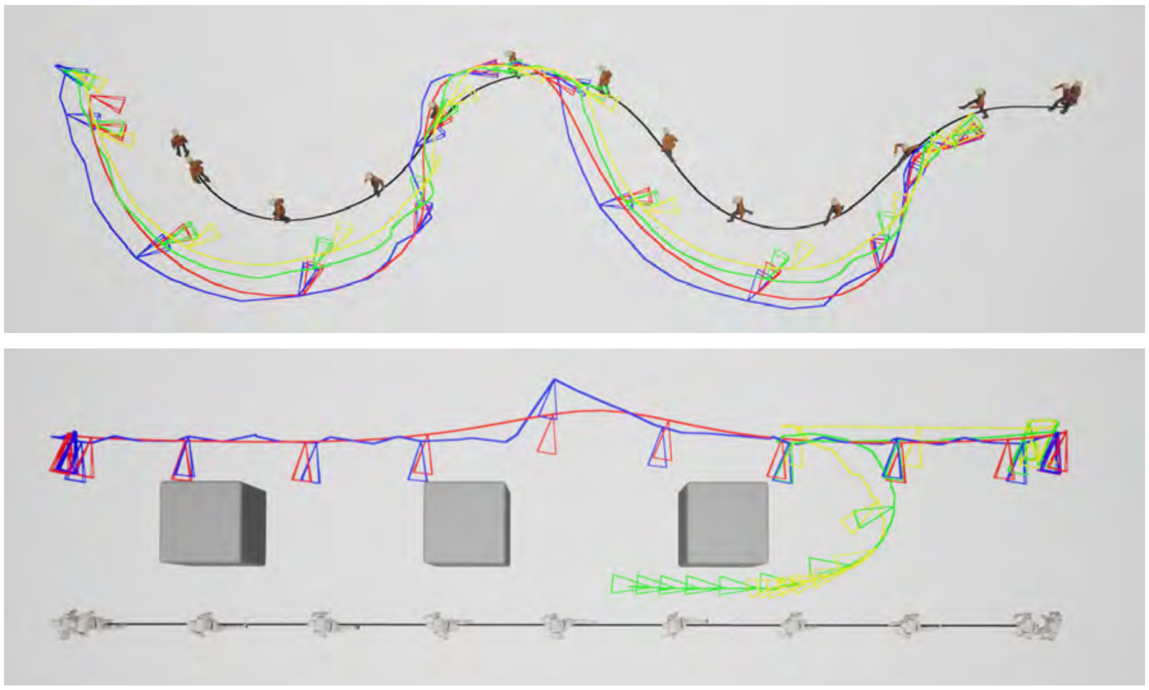


Figure 9.1: Visualizations of camera paths for different use cases. Overhead views of the camera paths computed for a simple moving scene (top) and a more complex scene where the camera's optimal path is blocked by a number of pillars (bottom). Color key: Green is unscripted-unsmoothed, yellow is unscripted-smoothed, red is scripted-unsmoothed, and blue is scripted-smoothed, while black is the path of the character.

each frame, with a warm start from the previous frame's computed camera settings.

2. Unscripted with smoothing: The system computes a camera pose for each frame online with the display of the frame, but with limited future knowledge allowing for a consideration of path smoothness a small interval into the future. Like the unscripted use case, this begins with a gradient based optimization for each keyframe individually, followed by smoothed gradient optimization with smoothing over all keyframes.

3. Scripted: Each frame is first optimized with a stochastic optimization approach (e.g., simulated annealing, PSO), the result of which is used to warm start a gradient based optimization (e.g., gradient descent, backtracking line search).

4. Scripted with smoothing: Results from scripted simulation are run through the implicit smoothing gradient descent of equation 4.13.

The first scene features a character running around an open area, where the objective was configured for the camera to follow the character from the front, following the recipe in Table 7.1. The path of the camera, as shown in the top image of Figure 9.1, is similar in all use cases tested.

An example of a scene in which there are noticeable dissimilarities between use cases is visualized in the bottom image of Figure 9.1. In this scene, a character runs past a set of pillars while the camera follows him with a long lens in profile. As the optimal camera path for the relative angle objective becomes occluded by the pillars, the system must compromise between the occlusion objective and the relative angle objective. As shown in Figure 9.1 (bottom), the paths of the various use cases vary much more significantly, and each suffers from its own particular issue.

The unscripted-unsmoothed strategy becomes stuck when it comes to the first occlusion boundary, while the unscripted-smoothed strategy manages to find its way around the first occlusion to trail the actor from a distance. While the latter solution seems intuitively better, the user has not explicitly specified which is preferable.

Meanwhile, both of the scripted strategies attempt to balance more evenly between all objectives, resorting to jagged solutions that momentarily provide an occluded the view of the actor. However, these solutions represent a technically fairer compromise between the desired shot objectives.

Additionally, the effects of varying the smoothness on a camera path are visualized in Figure 9.2. Note that, while the camera path becomes smoother as the smoothing weights are raised, that smoothness is not uniform across the entire path, with some keyframes having more abruptly turning curves where the objective is strongest. This ability to compromise between smoothness and objective values sets my method apart from alternate approaches that smooth as a behavior-incognizant post-processing operation.

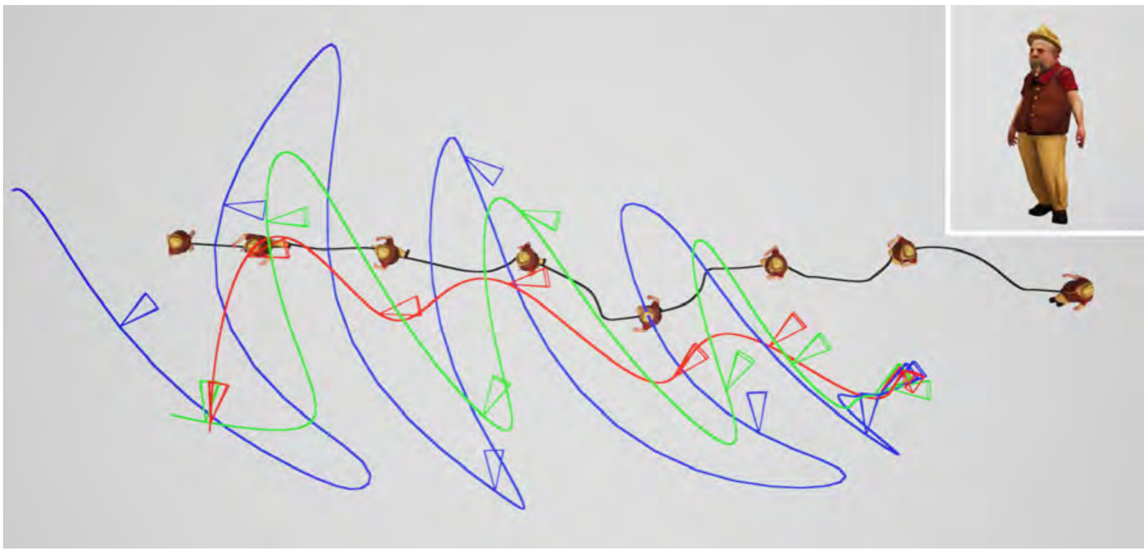


Figure 9.2: Visualization of camera paths resulting from varying smoothness parameters. The black path corresponds to the path of the actor, who turns left and right while walking straight. The blue, green, and red paths correspond to camera paths that achieve compositions similar to the image inset in the upper right corner. While all paths have a smoothing coefficient of $\alpha = 0$, the blue path has $\beta = 0.05$, the green path has $\beta = 0.5$, and the red path has $\beta = 5$.

9.3 Manually defined objective functions

To demonstrate the controllability of the system with manually defined objective functions, several example scenes have had camera control specified with a variety of different settings to the recipe described in Table 7.1, with paths visualized in Figures 9.3 and 9.4. Note the direct controllability in changes to shot size and angle demonstrated in these paths, with other parameters being similarly controllable.

Additionally, an example scene has been explored using the two-person cinematographic objective recipe described in Table 7.2, as visualized in Figure 9.5. Note that the system is capable of varying all of the same properties in this two-person scene as are parameterized in the recipe, including relative angle, shot size, and smoothness, across the same variety of use cases as were demonstrated for single person scenes.

A useful visualization for examining objective functions in this context takes the form of a contour plot, such as in Figure 9.6. Note that the objective function varies

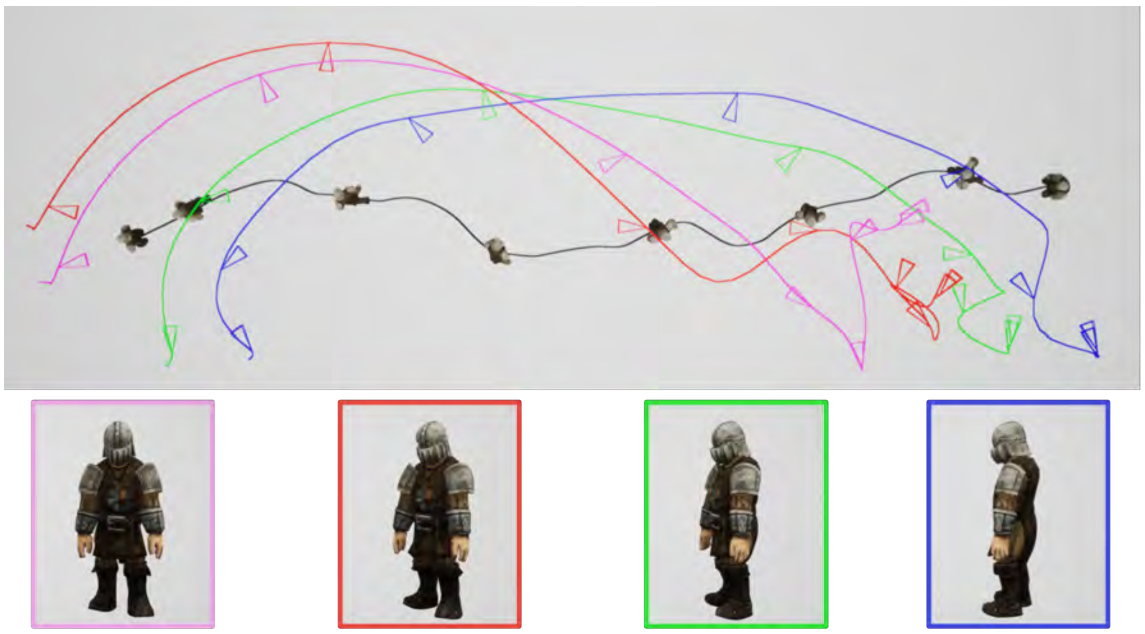


Figure 9.3: Visualization of camera paths resulting from different choices of horizontal relative angle. The black path in corresponds to the path of an actor running from the right side of the scene to the left, turning as he runs. The pink, red, green, and blue paths correspond to camera paths with a horizontal relative angle of 0°, 30°, 60°, and 90°respectively, as illustrated in the correspondingly colored insets below. Note that no line of axis constraint was included in this experiment, resulting in compositions that may switch between the character looking left or right over time.

meaningfully along all axes plotted, allowing for significant control authority of both the position and orientation of the camera.

9.4 Machine Learning defined objective functions

We can also briefly examine the capabilities of objective functions defined through the machine learning exercises described in Chapter 8. Unfortunately, as described in that section, expressivity is severely limited, with the latent control space only allowing for specification of the most statistically significant features in the dataset.

For example, a contour plot is displayed in Figure 9.7, which is probably best understood in comparison to Figure 9.6. While the MDL model displays complex rotational control authority, the positional plot shows minimal evidence for support for behaviors

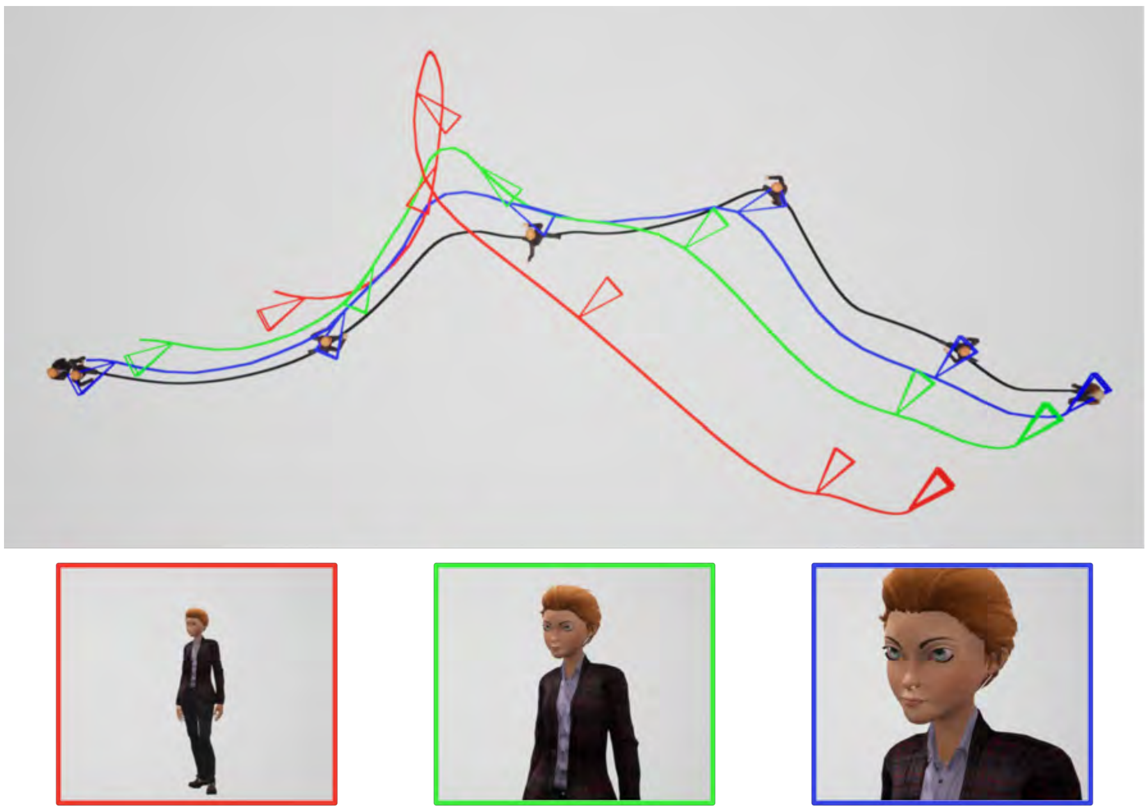


Figure 9.4: Visualization of camera paths resulting from different choices of shot size. The black path corresponds to the path of an actress walking from right to left through the scene, turning slightly as she moves. The red, green, and blue paths correspond to wide, medium, and close-up compositions, as illustrated in the correspondingly colored illustrations below.

like relative angles. Worse still, perturbing the inputs slightly does not seem to allow for significant changes in prioritization between what the manual objectives would consider distinct compositional behaviors, severely limiting the expressivity of this model.

As a result, camera paths with these objectives tend to appear like those shown in Figure 9.8, with the camera conforming shot size and frame position behaviors, but not incorporating much if any relative angle behavior. As a result, the camera has a tendency to follow behind the path of the actor, taking the shortest path that allows for the desired shot size.

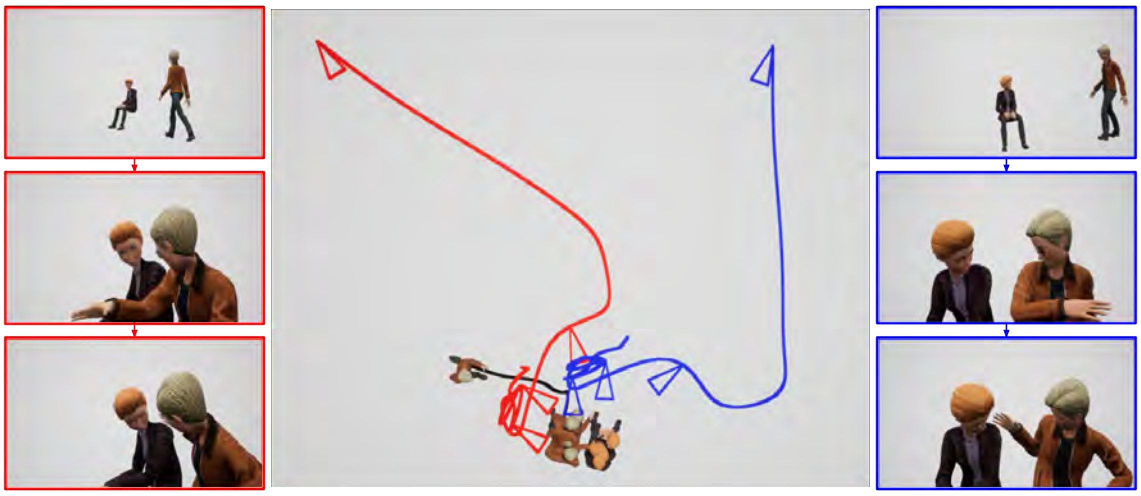


Figure 9.5: Visualization of camera paths for a scene with two people. In this scene, the glasses wearing actor walks over and sits next to the actress, where they have a short conversation. The black path corresponds to the path of the actor as he walks the few feet before sitting, while the actress remains in the same seat throughout the scene. Two camera paths were computed with the objective function described in Table 7.2 to begin with a wide shot and move into a medium-close-up. However, the horizontal relative angle of the two camera paths are different, with the blue path showing the two characters from perpendicular to the axis between them, and the red path showing the two characters from a $\sim 30°$angle towards the actress.

9.5 Speed

Measuring the speed of the system is difficult as it entirely depends on the thresholds used for terminating optimization, as well as the complexity of the objective functions being optimized. There is a tradeoff between processor time and solution quality, so a user willing to endure slower performance can set a higher bar for numerical quality. All timing measurements were taken on a desktop computer with a 4.20 GHz processor and an Nvidia GTX 1080Ti graphics card running my system integrated with `Unreal Engine 4.25`.[126]

For most of the *scripted* scenes tested, a satisfactory unsmoothed camera path could

[126]The core of the software is quite modular, with the only portions of code directly integrated with UE4 being isolated to a separately compilable software module. This portion of the system should be relatively easy to integrate with other engines, such as UE5, Unity, Maya, or Blender, but I have not yet attempted to do so.

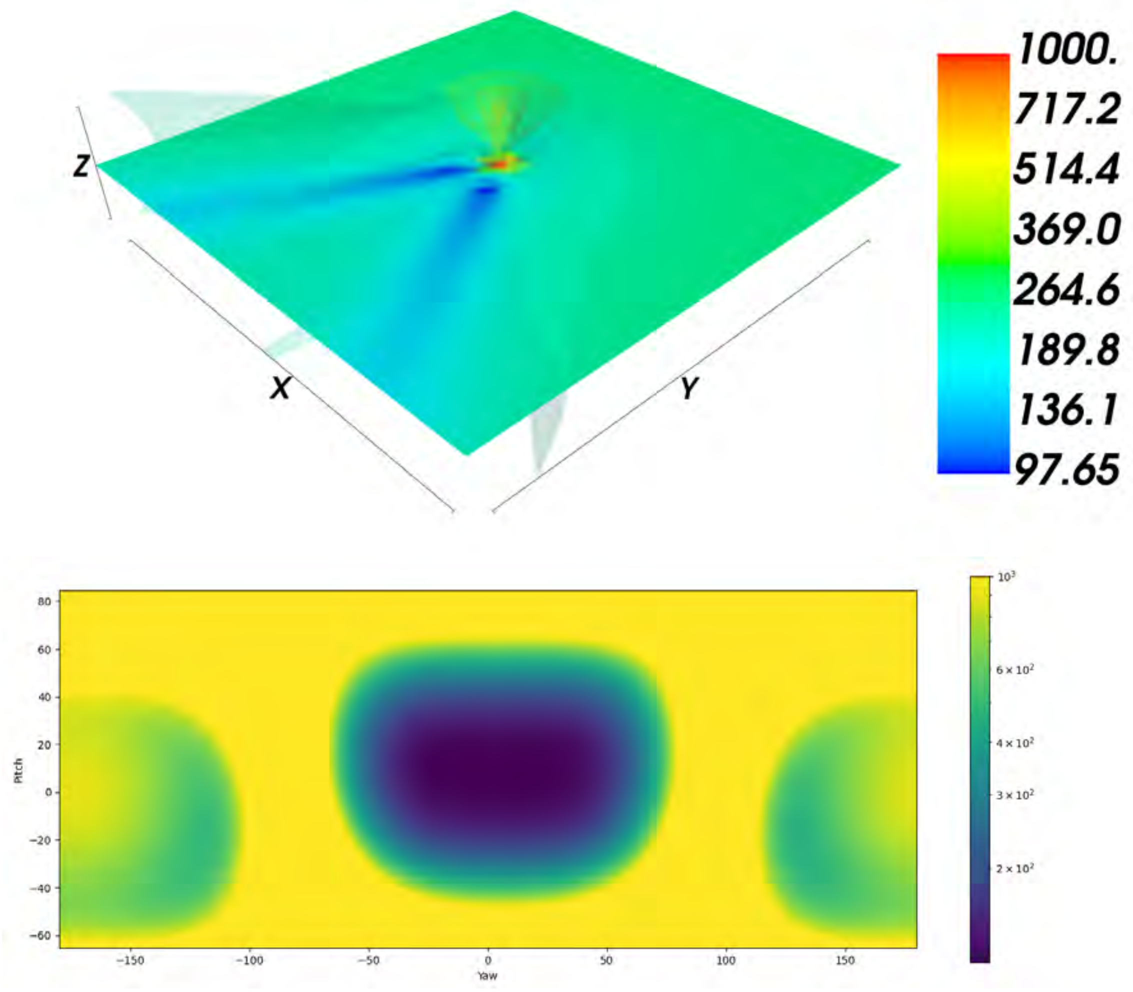
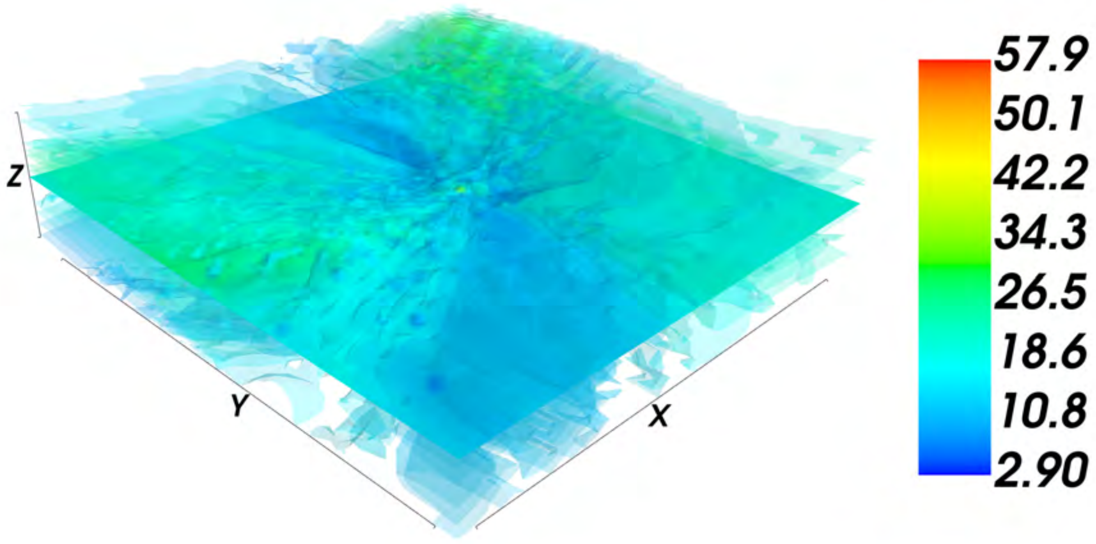


Figure 9.6: Plots of objective function values for the recipe from Table 7.1. These images plot the objective function values for the recipe from table 7.1, parameterized for a wide shot size and with a horizontal relative angle of 30°. The upper image plots the optimal value for each potential camera position in a fixed 3D volume, with a search allowed over camera yaw and pitch. The lower image plots the objective value over a range of yaw and pitch values from a fixed position.



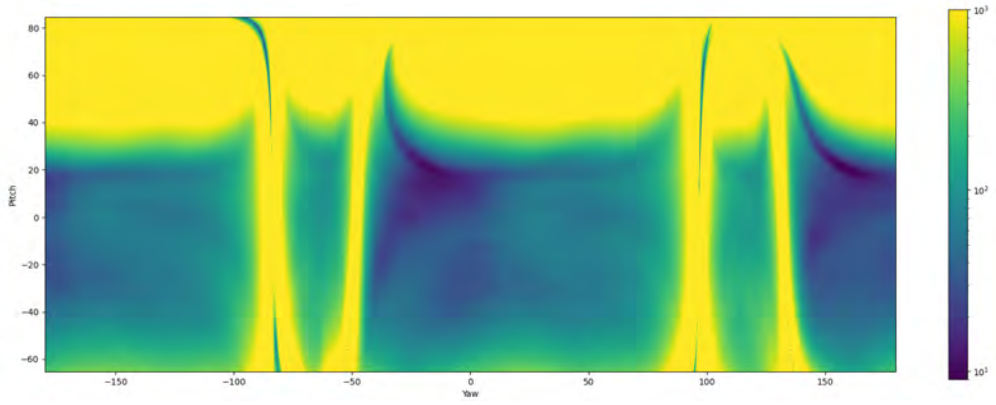


Figure 9.7: Plots of objective function values for a sample parameterization of machine learning derived MDL objective function. The upper image plots the optimal value for each potential camera position in a fixed 3D volume, with a search allowed over camera yaw and pitch. The lower image plots the objective value over a range of yaw and pitch values from a fixed position. Note that the curving bands of high values correspond to orientations that would project points of interest on the actor to near-infinite values on the image plane.

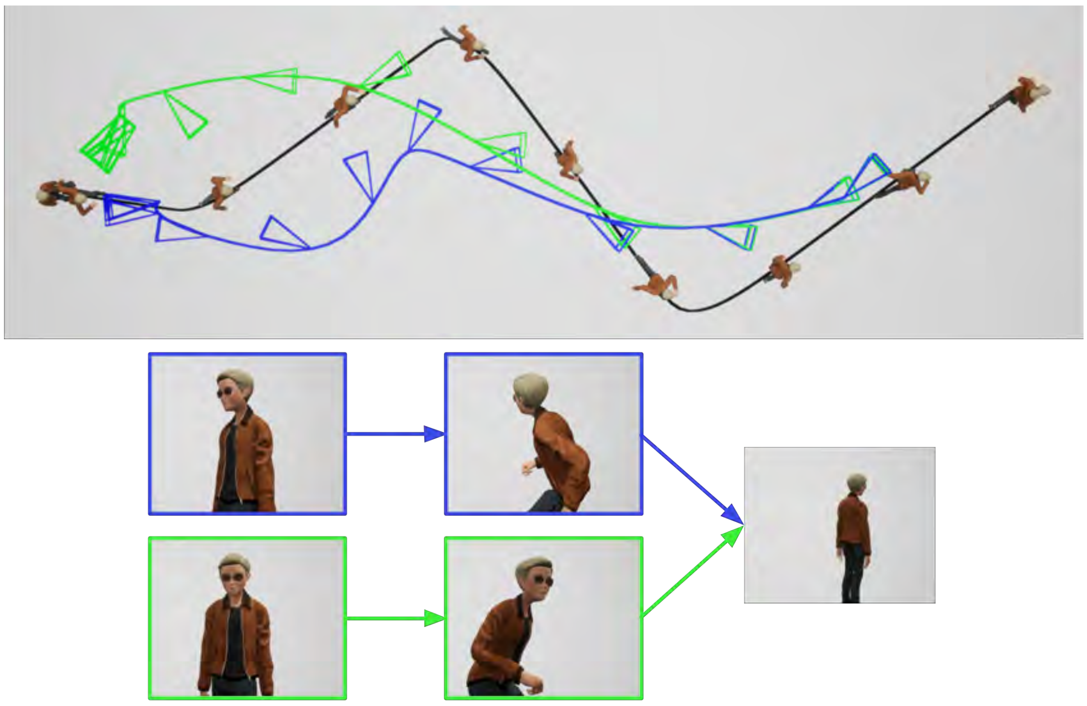


Figure 9.8: Visualization of camera paths resulting from machine learning constructed objective function. The black path corresponds to the path the actor takes as he runs from the left to the right of the scene. The green and blue paths correspond to camera paths computed using the same vector input to the MDL model, but with different starting search positions. Sample frames from the green and blue paths are illustrated in the correspondingly colored frames inset below. Note that while these two paths begin differently, they eventually converge to the same path following the actor, as the final inset frame shows.

Scenario	Smoothing	Stochastic Iterations	Gradient Iterations	Time/Keyframe
Scripted	No	3K	2K	1.15s
	Yes	3K	1K w/o smoothing, 1K w/ smoothing	1.75s
Unscripted	No	0	100	0.016s (60 FPS)
	Yes	0	10 w/o smoothing (for each keyframe), 10 w/ smoothing (across all keyframes)	0.04s (25 FPS)

Table 9.1: Summary of Average Computational Speed Across Different Use Cases

be found using approximately 3,000 iterations of simulated annealing, followed by approximately 2,000 iterations of gradient descent, taking about 1.15 seconds per keyframe. With smoothing, the same two-phase optimization can be used to roughly locally optimize using about 50–75% of the iterations required without smoothing, followed by 1,000–1,200 iterations of gradient descent with smoothing at a total cost of about 1.75 seconds per keyframe.

In *unscripted* experiments, the unsmoothed algorithm was subjectively found to be capable of finding satisfactory solutions using 100 optimization iterations of gradient descent at an average of 50-60 frames per second, depending on the particular objective function used. The smoothed algorithm was found to produce a subjectively satisfactory solution—given the prediction of 4 future keyframes with $t = 0.25\,\mathrm{sec}$ and with 10 iterations of gradient descent to initialize each keyframe, followed by another 10 iterations of active contour model implicit gradient descent—at an average of 25 frames per second, which is sufficient to support real-time usage.

However, achieving these speeds for each use case requires that the program constructed must first be analyzed and optimized by the tools described in Sections 5.3, which often requires a nontrivial computation time. Note that this static analysis and program optimization is only performed once, before the program begins attempting to compute camera positions at any keyframes, somewhat analogously to a compiler producing machine code. For unscripted unsmoothed use cases, this optimization takes less than 1.5 seconds, while unscripted smoothed scenarios can require 10-20 seconds of preprocessing optimization depending on the objective function used. Scripted use cases, both with and without smoothing, tend to require more time for preprocessing optimization, sometimes as much as 30 seconds. However, in all use cases, running the preprocessing program optimization provides a significant 30-60% speed improvement, making the computational cost worthwhile in the overwhelming majority of cases.

As discussed in Section 5.5, there are several engineering improvements that could significantly improve system speed. The current system has been engineered to be only as fast as necessary to support the experiments run, not maximally optimized for speed or efficiency. Consequently, these speed statistics should be interpreted as how fast the system needs to be to support these use cases, rather than an upper bound on the speed achievable with the techniques described in this text.

CHAPTER 10

Conclusion

10.1 Contributions

This thesis has explored answers to the question: Is it possible to engineer a software system to provide expressive, controllable, and efficient vector-based virtual cinematography automation for online and offline applications, with and without future knowledge?

The results shown in Chapter 9 demonstrate that the mathematical techniques and software proposed in Chapters 4 and 5 can be used to achieve a variety of declarative virtual cinematography applications with practicable efficiency. In addition, different approaches to providing more controllable and expressive interfaces to the system were explored, both as a library of easily combinable objective functions in Chapter 7 and as a novel machine learning model trained on data from feature films in Chapters 6 and 8.

Along the way, a novel taxonomy for understanding the diverse body of related virtual cinematography research was introduced in Chapter 3, a new dataset of human motion in feature films was collected as discussed in Chapter 6, and an unprecedented type of programming language modeled system for declarative virtual cinematography was proposed in Chapter 5, among other significant contributions discussed throughout the remainder of the text.

10.2 Future Work

Nevertheless, the techniques proposed, tools built, data collected, and results presented thus far leave a considerable number of areas of virtual cinematography open for future research. Some of these potential future work directions were previously discussed, such as the probability of computational speed improvements in Chapter 5, the possibility of estimating camera motion in a global reference frame in Chapter 6, and the practicability of alternative machine learning models to circumvent the limitations of the models explored here, among others. However, several areas of future virtual cinematography research remain to be discussed.

One of the most fascinating and challenging questions that to our knowledge has not been addressed, rests in attempting to quantify the usability of virtual cinematography systems. While abstract design principles, such as controllability and expressivity, are intuitively appealing in theory, attempting to measure the usability of systems with differing behavioral capabilities, interface paradigms, or algorithmic strategies for real users in real environments presents an enormous challenge. While running such an experiment might seem as simple as asking volunteers to use different virtual cinematography systems to attempt to achieve specific cinematographic goals and comparing their responses, a variety of prerequisite questions make such a test difficult. What types of use cases and cinematographic goals should be sampled, and how should user responses be measured? What types of end-users should such an experiment attempt to model, and how should volunteers be selected or directed to represent them? How should differences between differing cinematographic sensibilities among volunteers be weighted against their responses to using each system? Questions such as these are difficult to answer and represent a largely unexplored area ripe for future research.[127]

Another exciting direction for future research lies in attempting to automatically control not only the camera, but also other elements of the scene and high level behaviors. As discussed in Chapter 3, a variety of tasks are closely related to virtual cinematography, such as layout synthesis and character animation. Only limited work has been published that discusses how to combine virtual cinematography with such tasks, commonly with highly specific applications or behavioral properties in mind. Would it be possible to

[127]A good place to start on the subject is Hösl (2019), as well as many of the works referenced therein.

extend tools like those proposed here to automatically and generically control, or adjust the controls of, cameras, actors, and props together to achieve desired cinematographic compositions?

However, perhaps the most enticing possibility for future research rests yet beyond these well explored boundaries. Would it be possible to first generate a story, then determine how to layout the props for each scene, animate the actors, and control the camera, all to evoke the emotion in the generated story? This represents a truly daunting task requiring not only virtual cinematography, layout synthesis, and character animation, but also an unprecedented ability to automate complex natural language generation and processing tasks. Perhaps one day in the future, more advanced forms of tools discussed in this work may be capable of generating entire feature films based on each viewer's personal preferences, granting each of us a truly unique entertainment experience at the touch of a button.

APPENDIX A

Reverse Accumulating Autodiff Rules

In formal notation, the general strategy described in Chapter 5.4.2 can be expressed as a single transformation,

$$\delta_r(e) = x_{rd} = []; \alpha(e); \overline{e} = 1 \; ; \texttt{execr}(x_{rd}).$$

Here, the $\alpha(e)$ transformation depends upon the specific type of term, with

$$\begin{aligned}
\alpha(x) =& x, \\
\alpha(n) =& n, \\
\alpha(\texttt{peekv}(x)) =& \texttt{pushr}(x_{rd}, x_{temp} = \texttt{peekv}(\overline{x}) \; ; \texttt{popv}(\overline{x}) \; ; \texttt{pushv}(\overline{x}, x_{temp} + \overline{e}) \; ; \\
& \overline{e} = 0) : \texttt{peekv}(x), \\
\alpha(e) =& \texttt{pushr}(x_{rd}, \texttt{popv}(x_e) \; ; \overline{e} = 0) \; ; \texttt{pushv}(x_e, \beta(e)) \; ; \\
& \texttt{pushr}(x_{rd}, \theta(e)) : \texttt{peekv}(x_e) \text{ for all other } e \text{ not already listed}, \\
\alpha(x = e) =& \texttt{pushv}(x', x) \; ; (x = \alpha(e)) \; ; \texttt{pushr}(x_{rd}, (\overline{e} \mathrel{+}= \overline{x}) \; ; (\overline{x} = 0) \; ; \\
& (x = \texttt{peekv}(x')) \; ; \texttt{popv}(x')), \\
\alpha(\texttt{pushv}(x, e)) =& \texttt{pushv}(x, \alpha(e)) \; ; \texttt{pushr}(x_{rd}, \overline{e} \mathrel{+}= \texttt{peekv}(\overline{x}) \; ; \texttt{popv}(\overline{x}) \; ; \texttt{popv}(x)), \\
\alpha(\texttt{popv}(x)) =& \texttt{pushr}(x_{rd}, \texttt{pushv}(x, \texttt{peekv}(x')) \; ; \texttt{popv}(x') \; ; \texttt{pushv}(\overline{x}, 0)) \; ; \\
& \texttt{pushv}(x', \texttt{peekv}(x)) \; ; \texttt{popv}(x), \text{ and} \\
\alpha(t) =& \beta(t) \text{ for all } t \text{ not already listed, where} \\
\beta(t) =& \text{clone of } t \text{ where every child } t_c \text{ has been replaced with } \alpha(t_c).
\end{aligned}$$

The critical θ transformations, within which derivative data is passed between adjoint variables, also vary by expression type, but generally follow the standard calculus rules for simple differentiation.

e	$\theta(e)$
n	`skip`
x	`skip`
$s : e_s$	$\overline{e_s} \mathrel{+}= \overline{e}$
$e_1 + e_2$	$\overline{e_1} \mathrel{+}= \overline{e}\ ;\ \overline{e_2} \mathrel{+}= \overline{e}$
$e_1 - e_2$	$\overline{e_1} \mathrel{+}= \overline{e}\ ;\ \overline{e_2} \mathrel{+}= 0 - \overline{e}$
$e_1 \cdot e_2$	$\overline{e_1} \mathrel{+}= \overline{e} \cdot \gamma(e_2)\ ;\ \overline{e_2} \mathrel{+}= \gamma(e_1) \cdot \overline{e}$
$\frac{e_1}{e_2}$	$\overline{e_1} \mathrel{+}= \frac{\overline{e}}{\gamma(e_2)}\ ;\ \overline{e_2} \mathrel{+}= 0 - \frac{\gamma(e_1)\cdot\overline{e}}{\gamma(e_2)^2}$
e_s^n	$\overline{e_s} \mathrel{+}= n \cdot \gamma(e)^{n-1} \cdot \overline{e}$
$f(e_s)$	$\overline{e_s} \mathrel{+}= \frac{f(\gamma(e_s)+\overline{e})-f(\gamma(e_s))}{\overline{e}}$

Here, $\gamma(e)$ is an additional utility transformation that accesses the value of the expression at that moment of reversed operation order. Additionally, the $\mathrel{+}=$ operation is shorthand for incrementing the relevant variable by the amount specified. Formal rules for both using the variables already defined can be expressed as follows.

$$\gamma(n) = n \qquad (\overline{n} \mathrel{+}= e) = (\texttt{skip})$$

$$\gamma(x) = x \qquad (\overline{x} \mathrel{+}= e) = (\overline{x} = \overline{x} + e)$$

$$\gamma(e) = \texttt{peekv}(x_e) \qquad (\overline{e_1} \mathrel{+}= e_2) = (\overline{e_1} = \overline{e_1} + e_2)$$

(2011). Driver: San Francisco. [PC CD-ROM]. 39

(2015). Grand theft auto v credits - windows. https://www.mobygames.com/game/windows/grand-theft-auto-v/credits. Accessed: 2021-20-04. 54

(2018). Red dead redemption ii credits - playstation 4. https://www.mobygames.com/game/playstation-4/red-dead-redemption-ii/credits. Accessed: 2021-20-04. 54

(2022). Carnegie mellon university motion capture database. [Database]. http://http://mocap.cs.cmu.edu/. 207

(2022). Cinemetrics. [Database]. http://www.cinemetrics.lv/index.php. 90

Abdullah, R., Christie, M., Schofield, G., Lino, C., and Olivier, P. (2011). Advanced composition in virtual camera control. In *Smart Graphics*, pages 13–24. Springer, Berlin. 31, 75, 182

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: principles, techniques and tools*. Addison Wesley. 121, 129

Akenine-Möller, T., Haines, E., Hoffman, N., Pesce, A., Iwanicki, M., and Hillaire, S. (2018). *Real-Time Rendering*. A K Peters/CRC Press, fourth edition. 66

American Film Institute (1998). AFI's 100 YEARS...100 MOVIES. https://www.afi.com/afis-100-years-100-movies/. 157

American Film Institute (2012). The 100 Greatest Films of All Time. https://www.bfi.org.uk/sight-and-sound/greatest-films-all-time. 157

André, E., Finkler, W., Graf, W., Rist, T., Schauder, A., and Wahlster, W. (1993). Wip: The automatic synthesis of multimodal presentations. In *IMI'91: Proceedings of the 1991 International Conference on Intelligent Multimedia Interfaces*. 70, 86

Andujar, C., Vazquez, P., and Fairen, M. (2004). Way-finder: guided tours through complex walkthrough models. *Computer Graphics Forum*, 23(3):499–508. 71

Arijon, D. (1991). *Grammar of the film language*. Silman-James Press; Distributed by Samuel French Trade, 1st silman-james press ed edition. 29, 41

Ashtari, A., Stevšić, S., Nägeli, T., Bazin, J.-C., and Hilliges, O. (2020). Capturing subjective first-person view shots with drones for automated cinematography. *ACM Transactions on Graphics*, 39(5):1–14. 41

Bain, M., Nagrani, A., Brown, A., and Zisserman, A. (2020). Condensed movies: Story based retrieval with contextual embeddings. *arXiv*. 90, 91

Bares, W., McDermott, S., Boudreaux, C., and Thainimit, S. (2000a). Virtual 3D camera composition from frame constraints. In *Proc. 8th ACM International Conference on Multimedia*, pages 177–186. ACM Press. 43, 181

Bares, W. H., Gregoire, J. P., and Lester, J. C. (1998a). Realtime constraint-based cinematography for complex interactive 3d worlds. In *IAAI-98 Proceedings*. 37, 43, 44, 58

Bares, W. H. and Lester, J. C. (1997). Realtime generation of customized 3d animated explanations for knowledge-based learning environments. *AAAI-97 Proceedings*. 37

Bares, W. H. and Lester, J. C. (1999). Intelligent multi-shot visualization interfaces for dynamic 3d worlds. In *Proceedings of the 4th international conference on Intelligent user interfaces - IUI '99*. ACM Press. 57, 58

Bares, W. H., Thainimit, S., and McDermott, S. (2000b). A model for constraint-based camera planning. In *Proceedings of AAAI spring symposium on smart graphics*. From: AAAI Technical Report SS-00-04. Compilation copyright © 2000, AAAI (www.aaai.org). All rights reserved. 57, 71

Bares, W. H., Zettlemoyer, L. S., Rodriguez, D. W., and Lester, J. C. (1998b). Task-sensitive cinematography interfacesfor interactive 3d learning environments. In *Proceedings of the 3rd international conference on Intelligent user interfaces - IUI '98*. ACM Press. 57

Beckhaus, S. (2002). *Dynamic Potential Fields for Guided Exploration in Virtual Environments*. PhD thesis, Otto von Guericke University Magdeburg. 75

Blinn, J. (1988). Where am I? What am I looking at? *IEEE Computer Graphics and Applications*, 8(4):76–81. 31, 36, 37, 39, 42, 46, 52, 66, 67, 69, 88

Bonatti, R., Bucker, A., Scherer, S., Mukadam, M., and Hodgins, J. (2021). Batteries, camera, action! learning a semantic control space for expressive robot cinematography. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 33, 68

Bonatti, R., Wang, W., Ho, C., Ahuja, A., Gschwindt, M., Camci, E., Kayacan, E., Choudhury, S., and Scherer, S. (2020a). Autonomous aerial cinematography in unstructured environments with learned artistic decision-making. *Journal of Field Robotics*, 37(4):606–641. 75, 77

Bonatti, R., Zhang, Y., Choudhury, S., Wang, W., and Scherer, S. (2020b). Autonomous drone cinematographer: Using artistic principles to create smooth, safe, occlusion-free trajectories for aerial filming. In *Springer Proceedings in Advanced Robotics*, pages 119–129. Springer International Publishing. 75, 77

Bonneel, N., Coeurjolly, D., Digne, J., and Mellado, N. (2020). Code replicability in computer graphics. *ACM Transactions on Graphics*, 39(4). 88

Bowen, C. J. and Thompson, R. (2013). *Grammar of the Shot*. Taylor & Francis. 23, 24

Brown, B. (2013). *Cinematography theory and practice : image making for cinematographers and directors*. Taylor & Francis. 10, 21, 25, 26, 27

Bucker, A., Bonatti, R., and Scherer, S. (2021). Do you see what i see? coordinating multiple aerial cameras for robot cinematography. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 78

Burelli, P. (2012). *Interactive Virtual Cinematography*. PhD thesis, IT University of Copenhagen. 75

Burelli, P., Gaspero, L. D., Ermetici, A., and Ranon, R. (2008). Virtual camera composition with particle swarm optimization. In *Smart Graphics*, pages 130–141. Springer, Berlin. 44, 59, 75, 182

Burg, L., Lino, C., and Christie, M. (2020). Real-time anticipation of occlusions for automated camera control in toric space. *Computer Graphics Forum*, 39(2):523–533. 64

Burtnyk, N., Khan, A., Fitzmaurice, G., Balakrishnan, R., and Kurtenbach, G. (2002). Stylecam: Interactive stylized 3d navigationusing integrated spatial & temporal controls. In *Proceedings of the 15th annual ACM symposium on User interface software and technology - UIST '02*. ACM Press. 54, 68

Butz, A. (1997). Anymation with cathi. In *AAAI'97/IAAI'97: Proceedings of the fourteenth national conference on artificial intelligence and ninth conference on Innovative applications of artificial intelligence*, pages 57–962. 86

Cao, Z., Hidalgo, G., Simon, T., Wei, S.-E., and Sheikh, Y. (2019). Openpose: Realtime multi-person 2d pose estimation using part affinity fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43:172–186. 164

Cao, Z., Simon, T., Wei, S.-E., and Sheikh, Y. (2017). Realtime multi-person 2d pose estimation using part affinity fields. In *CVPR*. 164

Castellano, B. (2020). Pyscenedetect: Video scene cut detection and analysis tool (version 0.5.2). [Software]. https://github.com/Breakthrough/PySceneDetect. 161

Chaudhuri, P., Kalra, P., and Banerjee, S. (2007). *View-Dependent Character Animation*. Springer. 85

Chen, C., Wang, O., Heinzle, S., Carr, P., Smolic, A., and Gross, M. (2013). Computational sports broadcasting: Automated director assistance for live sports. In *2013 IEEE International Conference on Multimedia and Expo (ICME)*. IEEE. 85, 91

Chen, J. and Carr, P. (2014). Autonomous camera systems: A survey. In *Workshops at the Twenty-Eighth AAAI Conference on Artificial Intelligence*. 78

Chen, J. and Carr, P. (2015). Mimicking human camera operators. In *2015 IEEE Winter Conference on Applications of Computer Vision*. IEEE. 68, 83, 91

Chen, J., Le, H. M., Carr, P., Yue, Y., and Little, J. J. (2016). Learning online smooth predictors for realtime camera planning using recurrent decision trees. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4688–4696. 68, 78, 91

Chen, M., Mountford, S. J., and Sellen, A. (1988). A study in interactive 3-d rotationusing 2-d control devices. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques - SIGGRAPH '88*. ACM Press. 67

Chiou, R. C. H., Kaufman, A. E., Liang, Z., Hong, L., and Achniotou, M. (1998). Interactive path planning for virtual endoscopy. In *1998 IEEE Nuclear Science Symposium Conference Record. 1998 IEEE Nuclear Science Symposium and Medical Imaging Conference (Cat. No.98CH36255)*. IEEE. 78

Christianson, D. B., Anderson, S. E., He, L.-w., Salesin, D., Weld, D. S., and Cohen, M. F. (1996). Declarative camera control for automatic cinematography. In *AAAI*. 40, 60

Christie, M. and Languénou, E. (2003). A constraint-based approach to camera path planning. In *Smart Graphics*, pages 172–181. Springer Berlin Heidelberg. 68, 73

Christie, M., Languénou, E., and Granvilliers, L. (2002). Modeling camera control with constrained hypertubes. In *Lecture Notes in Computer Science*, pages 618–632. Springer Berlin Heidelberg. 37, 43, 57, 68, 73

Christie, M., Lino, C., and Ronfard, R. (2012). Film editing for third person games and machinima. In *Workshop on Intelligent Cinematography and Editing*. 84

Christie, M., Machap, R., Normand, J.-M., Olivier, P., and Pickering, J. (2005). Virtual camera planning: A survey. In *Smart Graphics*, pages 40–52. Springer, Berlin. 31, 46

Christie, M. and Normand, J.-M. (2005). A semantic space partitioning approach to virtual camera composition. In *Computer Graphics Forum*, volume 24, pages 247–256. Wiley. 57

Christie, M., Olivier, P., and Normand, J.-M. (2008). Camera control in computer graphics. In *Computer Graphics Forum*, volume 27, pages 2197–2218. Wiley Online Library, Wiley. 31, 64, 69

Church, A. (1941). *The Calculi of Lambda Conversion*. Princeton University Press. 140

Cook, D. A. (2004). *A history of narrative film*. W.W. Norton. 28

Courant, R., Lino, C., Christie, M., and Kalogeiton, V. (2021). High-level features for movie style understanding. In *ICCV 2021 - Workshop on AI for Creative Video Editing and Understanding*. 91, 205

Courty, N. and Marchand, E. (2001). Computer animation: A new application for image-based visual servoing. In *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*. IEEE. 69, 79

Cousot, P. and Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. 129

Cutting, J. E., Brunick, K. L., DeLong, J. E., Iricinschi, C., and Candan, A. (2011). Quicker, faster, darker: Changes in hollywood film over 75 years. *i-Perception*, 2(6):569–576. 90, 162

Cutting, J. E., DeLong, J. E., and Nothelfer, C. E. (2010). Attention and the evolution of hollywood film. *Psychological Science*, 21(3):432–439. 90

Drucker, S. M. (1994). *Intelligent Camera Control for Graphical Environments*. PhD thesis, Massachusetts Institute of Technology. 77

Drucker, S. M. and Galyean, T. A. (1992). CINEMA: A system for procedural camera movements. In *I3D '92*. 55, 88

Drucker, S. M. and Zeltzer, D. (1994). Intelligent camera control in a virtual environment. In *Graphics Interface*. 59

Drucker, S. M. and Zeltzer, D. (1995). CamDroid: a system for implementing intelligent camera control. In *Proc. 1995 ACM Symposium on Interactive 3D Graphics*, pages 139–144. ACM Press. 77, 186

Effron, L. and Gowen, G. (2018). 'Jurassic Park' turns 25. `https://abcnews.go.com/Entertainment/jurassic-park-turns-25-scenes-moments-iconic-summer/story?id=55332468`. 172

Elson, D. K. and Riedl, M. O. (2007). A lightweight intelligent virtual cinematography system for machinima production. In *Proc. Artificial Intelligence and Interactive Digital Entertainment Conf.*, pages 8–13. 37, 41, 68, 85

Epic Games (2022). Unreal engine (version 4.25.4). [Software]. https://www.unrealengine.com. 54, 65, 70, 85, 113

Espiau, B., Chaumette, F., and Rives, P. (1993). A new approach to visual servoing in robotics. In *Geometric Reasoning for Perception and Action*, pages 106–136. Springer Berlin Heidelberg. 69, 79

Fang, C., Lin, Z., Měch, R., and Shen, X. (2014). Automatic image cropping using visual composition, boundary simplicity and content preservation models. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM. 83

Feiner, S. K. and Seligmann, D. D. (1992). Cutaways and ghosting: satisfying visibility constraints in dynamic 3d illustrations. *The Visual Computer*, 8(5-6):292–302. 58, 70, 86

Funge, J., Tu, X., and Terzopoulos, D. (1999). Cognitive modeling: Knowledge, reasoning and planning for intelligent characters. In *Proc. ACM SIGGRAPH*, pages 29–38. ACM Press. 57, 67, 72, 76

Gaddam, V. R., Eg, R., Langseth, R., Griwodz, C., and Halvorsen, P. (2015). The cameraman operating my virtual camera is artificial: Can the machine be as good as a human? *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 11(4):1–20. 83

Galvane, Q., Christie, M., Lino, C., and Ronfard, R. (2015a). Camera-on-rails: Automated computation of constrained camera paths. In *Proc. 8th ACM SIGGRAPH Conference on Motion in Games*, pages 151–157. ACM. 66, 191

Galvane, Q., Christie, M., Ronfard, R., Lim, C.-K., and Cani, M.-P. (2013). Steering behaviors for autonomous cameras. In *Proceedings of Motion on Games*. ACM. 44, 68, 75, 85

Galvane, Q., Ronfard, R., Lino, C., and Christie, M. (2015b). Continuity editing for 3d animation. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 84, 91

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns*. Addison-Wesley Professional. 132

Giors, J. (2004). The full spectrum warrior camera system. *Game Developer*. 44, 55, 68, 70

Gleicher, M. (1994). *A Differential Approach to Graphical Interaction*. PhD thesis, Carnegie Melon University. 89

Gleicher, M. and Witkin, A. (1992). Through-the-lens camera control. *Computer Graphics*, 26(2):331–140. (Proc. ACM SIGGRAPH ’92). 37, 60

Goi, M., editor (2013). *American cinematographer manual*. The ASC Press, tenth edition. 20

Goodfellow, I., Bengio, Y., and Courville, A. (2017). *Deep Learning*. MIT Press. 82

Graham, P. (2009). Doug mcilroy: Mccarthy presents lisp. `http://www.paulgraham.com/mcilroy.html`. Accessed: 2022-04-13. 140

Gu, C., Sun, C., Ross, D. A., Vondrick, C., Pantofaru, C., Li, Y., Vijayanarasimhan, S., Toderici, G., Ricco, S., Sukthankar, R., Schmid, C., and Malik, J. (2017). Ava: A video dataset of spatio-temporally localized atomic visual actions. *arXiv*. 90

Guennebaud, G., Jacob, B., et al. (2010). Eigen v3. http://eigen.tuxfamily.org. 113

Hagen, C. (2017). (correctly) averaging quaternions. [Software]. https://github.com/christophhagen/averaging-quaternions. 207

Haigh-Hutchinson, M. (2009). *Real-time cameras: a guide for game designers and developers*. Morgan Kaufmann. 31, 32, 39, 51, 55, 68

Halper, N., Helbing, R., and Strothotte, T. (2001). A camera engine for computer games: Managing the trade-off between constraint satisfaction and frame coherence. *Computer Graphics Forum*, 20(3):174–183. 72, 108

Halper, N. and Olivier, P. (2000). Camplan: A camera planning agent. In *Smart Graphics 2000 AAAI Spring Symposium*. 33, 75

Hassanien, A., Elgharib, M., Selim, A., Bae, S.-H., Hefeeda, M., and Matusik, W. (2017). Large-scale, fast and accurate shot boundary detection through spatio-temporal convolutional neural networks. *arXiv*. 161

He, L.-W., Cohen, M. F., and Salesin, D. H. (1996). The virtual cinematographer: A paradigm for automatic real-time camera control and directing. In *Proc. 23rd Annual Conference on Computer Graphics and Interactive Techniques*, pages 217–224. ACM, ACM Press. 40, 54, 68, 84, 88

Hong, L., Muraki, S., Kaufman, A., Bartz, D., and He, T. (1997). Virtual voyage: Interactive navigation in the human colon. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques - SIGGRAPH '97*. ACM Press. 78

Hösl, A. (2019). *Understanding and Designing for Control in Camera Operation*. PhD thesis, Ludwig Maximilian University of Munich. 221

Huang, C., Dang, Y., Chen, P., Yang, X., and Cheng, K.-T. (2019a). One-shot imitation filming of human motion videos. *arXiv*. 33

Huang, C., Dang, Y., Chen, P., Yang, X., and Cheng, K.-T. T. (2021). One-shot imitation drone filming of human motion videos. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–1. 33, 41, 66

Huang, C., Lin, C.-E., Yang, Z., Kong, Y., Chen, P., Yang, X., and Cheng, K.-T. (2019b). Learning to film from professional human motion videos. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. 33

Huang, H., Lischinski, D., Hao, Z., Gong, M., Christie, M., and Cohen-Or, D. (2016). Trip synopsis: 60km in 60sec. *Computer Graphics Forum*, 35(7):107–116. 77, 88

Huang, Q., Xiong, Y., Rao, A., Wang, J., and Lin, D. (2020). MovieNet: A holistic dataset for movie understanding. In *The European Conference on Computer Vision (ECCV)*, pages 709–727. Springer International Publishing. 90

Ierusalimschy, R. (2006). *Programming in lua*. Roberto Ierusalimschy. 113

Ionescu, C., Li, F., and Sminchisescu, C. (2011). Latent structured models for human pose estimation. In *International Conference on Computer Vision*. 207

Ionescu, C., Papava, D., Olaru, V., and Sminchisescu, C. (2014). Human3.6m: Large scale datasets and predictive methods for 3d human sensing in natural environments. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(7):1325–1339. 207

Jakob, W., Speierer, S., Roussel, N., and Vicini, D. (2022). DR.JIT. *ACM Transactions on Graphics*, 41(4):1–19. 81

Jardillier, F. and Languénou, E. (1998). Screen-space constraints for camera movements: The virtual cameraman. *Computer Graphics Forum*, 17(3):175–186. 73, 181

Jiang, H., Christie, M., Wang, X., Liu, L., Wang, B., and Chen, B. (2021). Camera keyframing with style and control. *ACM Trans. Graph.*, 40(6). 39, 41, 68, 88

Jiang, H., Wang, B., Wang, X., Christie, M., and Chen, B. (2020). Example-driven virtual cinematography by learning camera behaviors. *ACM Transactions on Graphics (TOG)*, 39(4):45–1. 39, 41, 53, 61, 66, 67, 68, 91, 195

Jovane, A., Louarn, A., and Christie, M. (2020). Topology-aware camera control for real-time applications. In *Motion, Interaction and Games*. ACM. 68

Kass, M., Witkin, A., and Terzopoulos, D. (1988). Snakes: Active contour models. *International Journal of Computer Vision*, 1(4):321–331. 102

Kato, H., Beker, D., Morariu, M., Ando, T., Matsuoka, T., Kehl, W., and Gaidon, A. (2020). Differentiable rendering: A survey. *arXiv preprint arXiv:2006.12057*. 80

Katz, S. D. (1991). *Film directing shot by shot*. Michael Wiese Productions in conjunction with Focal Press. 29

Kellnhofer, P., Recasens, A., Stent, S., Matusik, W., and Torralba, A. (2019). Gaze360: Physically unconstrained gaze estimation in the wild. In *IEEE International Conference on Computer Vision (ICCV)*. 165

Kessenich, J., Sellers, G., and Shreiner, D. (2016). *OpenGL Programming Guide The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V*. Addison-Wesley Professional. 66

King, D. E. (2009). Dlib-ml: A machine learning toolkit. *Journal of Machine Learning Research*, 10:1755–1758. 163

Kirchner, F. and Sinot, F.-R. (2007). Rule-based operational semantics for an imperative language. *Electronic Notes in Theoretical Computer Science*, 174(1):35–47. 117

Kmett, E. (2022). Haskell package ad: Automatic Differentiation. [Software]. https://hackage.haskell.org/package/ad. 150

Koyama, Y., Sato, I., Sakamoto, D., and Igarashi, T. (2017). Sequential line search for efficient visual design optimization by crowds. *ACM Transactions on Graphics*, 36(4):1–11. 82

Larson, R. and Edwards, B. H. (2009). *Calculus*. Cengage Learning. 145

Le Cun, Y. and Fogelman-Soulié, F. (1987). Modèles connexionnistes de l'apprentissage. *Intellectica*, 2(1):114–143. 140

Leake, M., Davis, A., Truong, A., and Agrawala, M. (2017). Computational video editing for dialogue-driven scenes. *ACM Trans. Graph.*, 36(4):130–1. 84

Lei, J., Yu, L., Bansal, M., and Berg, T. L. (2018). Tvqa: Localized, compositional video question answering. *arXiv*. 90

Lei, J., Yu, L., Berg, T. L., and Bansal, M. (2019). Tvqa+: Spatio-temporal grounding for video question answering. *arXiv*. 90

Li, J., Xu, K., Chaudhuri, S., Yumer, E., Zhang, H., and Guibas, L. (2017). Grass: Generative recursive autoencoders for shape structures. *ACM Transactions on Graphics*, 36(4):1–14. 82

Li, T.-M., Aittala, M., Durand, F., and Lehtinen, J. (2018). Differentiable monte carlo ray tracing through edge sampling. *ACM Transactions on Graphics*, 37(6):1–11. 81

Linnainmaa, S. (1970). The representation of the cumulative rounding error of an algorithm as a taylor expansion of the local rounding errors. Master's thesis, Master's Thesis (in Finnish), Univ. Helsinki. 140

Lino, C. (2015). Toward more effective viewpoint computation tools. *Eurographics Workshop on Intelligent Cinematography and Editing*. 37, 43, 59, 76

Lino, C. and Christie, M. (2012). Efficient composition for virtual camera control. In *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. The Eurographics Association. 36, 37, 40, 69

Lino, C. and Christie, M. (2015). Intuitive and efficient camera control with the toric space. *ACM Trans. Graph.*, 34(4):1–12. 40, 44, 53, 191

Lino, C., Christie, M., Lamarche, F., Schofield, G., and Olivier, P. (2010). A real-time cinematography system for interactive 3D environments. In *Proc. 2010 ACM SIGGRAPH/EG Symposium on Computer Animation*, pages 139–148. 40, 73

Lino, C., Christie, M., Ranon, R., and Bares, W. (2011). The director's lens: An intelligent assistant for virtual cinematography. In *Proceedings of the 19th ACM international conference on Multimedia - MM '11*. ACM Press. 73

Lino, C., Christie, M., Ranon, R., and Bares, W. (2013). A smart assistant for shooting virtual cinematography with motion-tracked cameras. In *Proceedings of the 19th ACM international conference on Multimedia - MM '11*. ACM Press. 73

Lino, C., Ronfard, R., Galvane, Q., and Gleicher, M. (2014). How do we evaluate the quality of computational editing systems? In *WICED@AAAI*, pages 35–39. 84

Litteneker, A. and Terzopoulos, D. (2017). Virtual cinematography using optimization and temporal smoothing. In *Proceedings of the Tenth International Conference on Motion in Games*, page 17. ACM. 59, 113, 144

Louarn, A., Christie, M., and Lamarche, F. (2018). Automated staging for virtual cinematography. In *Proceedings of the 11th Annual International Conference on Motion, Interaction, and Games*. ACM. 43, 86

Louarn, A., Galvane, Q., Lamarche, F., and Christie, M. (2020). An interactive staging-and-shooting solver for virtual cinematography. In *Motion, Interaction and Games*. ACM. 73, 86

Loubet, G., Holzschuch, N., and Jakob, W. (2019). Reparameterizing discontinuous integrands for differentiable rendering. *ACM Transactions on Graphics*, 38(6):1–14. 80

Mackinlay, J. D., Card, S. K., and Robertson, G. G. (1990). Rapid controlled movement through a virtual 3d workspace. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '90, pages 171–176, New York, NY, USA. Association for Computing Machinery. 67

Marchand, E. and Courty, N. (2000). Image-based virtual camera motion strategies. In *Graphics Interface Conference*. 69

Marchand, É. and Courty, N. (2002). Controlling a camera in a virtual environment. *The Visual Computer*, 18(1):1–19. 69

Marchand, E. and Hager, G. D. (1998). Dynamic sensor planning in visual servoing. In *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No.98CH36146)*. IEEE. 69, 79

Marin-Jimenez, M., Zisserman, A., and Ferrari, V. (2011). "Here's looking at you, kid." detecting people looking at each other in videos. In *British Machine Vision Conference*. 165

Markley, F. L., Cheng, Y., Crassidis, J. L., and Oshman, Y. (2007). Averaging quaternions. *Journal of Guidance, Control, and Dynamics*, 30(4):1193–1197. 207

Markoff, J. (1994). Gary kildall, 52, crucial player in computer development, dies. *The New York Times*. 126

Mehta, D., Sotnychenko, O., Mueller, F., Xu, W., Elgharib, M., Fua, P., Seidel, H.-P., Rhodin, H., Pons-Moll, G., and Theobalt, C. (2020). XNect: Real-time multi-person 3D motion capture with a single RGB camera. *ACM Transactions on Graphics*, 39(4). 166

Merabti, B., Christie, M., and Bouatouch, K. (2015). A virtual director using hidden markov models. *Computer Graphics Forum*, 35(8):51–67. 84

Metz, C. (1991). *Film language: A semiotics of the cinema.* University of Chicago Press. 29

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. In *International Conference on Learning Representations (ICLR).* 201

Mobbs, D., Weiskopf, N., Lau, H. C., Featherstone, E., Dolan, R. J., and Frith, C. D. (2006). The kuleshov effect: the influence of contextual framing on emotional attributions. *Social cognitive and affective neuroscience*, 1(2):95–106. 28

Moore, R. E. (1966). *Interval analysis.* Prentice-Hall. 125

Moss, R. (2018). How bad crediting hurts the game industry and muddles history. 54

Naumann, U. (2011). *The art of differentiating computer programs: an introduction to algorithmic differentiation.* SIAM. 142

O'Hara, K. (2020). Optimlib: a lightweight c++ library of numerical optimization methods for nonlinear functions (revision x). [Software]. https://github.com/kthohr/optim. 113

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*, volume 1, pages 22–30. 33, 44, 59, 64, 75

Oskam, T., Sumner, R. W., Thuerey, N., and Gross, M. (2009). Visibility transition planning for dynamic camera control. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation - SCA '09*. ACM Press. 72, 76

Osokin, D. (2019). Real-time 2d multi-person pose estimation on cpu: Lightweight openpose. 166

Osokin, D. and Ageeva, M. (2020). Real-time 3d multi-person pose estimation demo (version 0.5.2). [Software]. https://github.com/Daniil-Osokin/lightweight-human-pose-estimation-3d-demo.pytorch. 166

Padia, K., Bandara, K. H., and Healey, C. G. (2019). A system for generating storyline visualizations using hierarchical task network planning. *Computers & Graphics*, 78:64–75. 178

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc. 197

Patow, G. and Pueyo, X. (2003). A survey of inverse rendering problems. *Computer Graphics Forum*, 22(4):663–687. 80

Phillipson, G., Jolly, S., Sheikh, A., Mills, P., Stagg, R., and Evans, M. (2022). 'old school': An 8k multicamerashoot to create a dataset for computational cinematography. Technical report, British Broadcasting Corporation. 91

Pickering, J. H. (2002). *Intelligent Camera Planning for Computer Graphics*. PhD thesis, University of York. 31, 57

Pickering, J. H. and Olivier, P. (2003). Declarative camera planning roles and requirements. In Butz, A. et al., editors, *International Symposium on Smart Graphics*, volume 2733 of *LNCS*, pages 182–191. Springer, Springer. 57, 58, 66

Pierce, B. C. and Benjamin, C. (2002). *Types and programming languages*. MIT press. 116, 117

Plantinga, H. and Dyer, C. R. (1990). Visibility, occlusion, and the aspect graph. *International Journal of Computer Vision*, 5(2):137–160. 65

Poulin, P., Ratib, K., and Jacques, M. (1997). Sketching shadows and highlights to position lights. In *Proceedings Computer Graphics International*. IEEE. 80

Ranon, R. and Urli, T. (2014). Improving the efficiency of viewpoint composition. *IEEE Transactions on Visualization and Computer Graphics*, 20(5):795–807. 37, 44, 76, 88

Rao, A., Wang, J., Xu, L., Jiang, X., Huang, Q., Zhou, B., and Lin, D. (2020a). A unified framework for shot type classification based on subject centric lens. In *Computer Vision – ECCV 2020*, pages 17–34. Springer International Publishing. 91

Rao, A., Xu, L., Xiong, Y., Xu, G., Huang, Q., Zhou, B., and Lin, D. (2020b). A local-to-global approach to multi-modal movie scene segmentation. *arXiv*. 90

Rogers, S. (2014). *Level Up The Guide To Great Video Game Design*. Wiley, second edition. 55

Rogez, G., Weinzaepfel, P., and Schmid, C. (2017). LCR-Net: Localization-Classification-Regression for Human Pose. In *CVPR*, Honolulu, United States. 166

Rohrbach, A., Rohrbach, M., Tandon, N., and Schiele, B. (2015). A dataset for movie description. *arXiv*. 90

Ronfard, R. (2012). A review of film editing techniques for digital games. In *Workshop on Intelligent Cinematography and Editing*. 83

Ronfard, R. (2021). Film directing for computer games and animation. *Computer Graphics Forum*, 40(2):713–730. 87

Ronfard, R. and de Verdière, R. C. (2022). OpenKinoAI: A framework for intelligent cinematography and editing of live performances. *Leonardo*, pages 373–377. 83

Rosenbrock, H. H. (1960). An Automatic Method for Finding the Greatest or Least Value of a Function. *The Computer Journal*, 3(3):175–184. 117

Ruiz, N., Chong, E., and Rehg, J. M. (2018). Fine-grained head pose estimation without keypoints. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. 164

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088):533–536. 140

Russell, S. J., Norvig, P., and Davis, E. (2010). *Artificial intelligence: a modern approach*. Prentice Hall series in artificial intelligence. Prentice Hall, 3rd ed edition. 72, 74

Sack, W. and Davis, M. (1994). IDIC: Assembling video sequences from story plans and content annotations. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems MMCS-94*. IEEE. 84

Salt, B. (1974). Statistical style analysis of motion pictures. *Film Quarterly*, 28(1):13–22. 90

Salt, B. (2006). *Moving Into Pictures*. Starword. 90

Salt, B. (2009). *Film Style & Technology: History and Analysis*. Starword. 90

Sanokho, C. B., Desoche, C., Merabti, B., Li, T.-Y., and Christie, M. (2014). Camera motion graphs. *Eurographics/ ACM SIGGRAPH Symposium on Computer Animation*. 68

Schneider, P. and Eberly, D. H. (2002). *Geometric Tools for Computer Graphics (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann. 66

Seligmann, D. D. and Feiner, S. (1991). Automated generation and of intent-based and 3d illustrations. *ACM SIGGRAPH Computer Graphics*, 25(4):123–132. 57, 58, 70, 86

Setlur, V., Takagi, S., Raskar, R., Gleicher, M., and Gooch, B. (2005). Automatic image retargeting. In *Proceedings of the 4th international conference on Mobile and ubiquitous multimedia - MUM '05*, pages 59–68. ACM Press. 82, 83

Shoemake, K. (1992). ARCBALL: A user interface for specifying three-dimensional orientation using a mouse. In *Graphics Interface*, volume 92, pages 151–156. 36, 61

Shuai, Q., Geng, C., Fang, Q., Peng, S., Shen, W., Zhou, X., and Bao, H. (2022). Novel view synthesis of human interactions from sparse multi-view videos. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Proceedings*. ACM. 82

Simon, T., Joo, H., Matthews, I., and Sheikh, Y. (2017). Hand keypoint detection in single images using multiview bootstrapping. In *CVPR*. 164

Solomon, J. (2015). *Numerical Algorithms: Methods for Computer Vision, Machine Learning, and Graphics*. CRC Press. 72, 74, 97

Sun, T., Barron, J. T., Tsai, Y.-T., Xu, Z., Yu, X., Fyffe, G., Rhemann, C., Busch, J., Debevec, P., and Ramamoorthi, R. (2019). Single image portrait relighting. *ACM Transactions on Graphics*, 38(4):1–12. 82

Sun, Y., Chao, Q., and Li, B. (2022). Synopses of movie narratives: a video-language dataset for story understanding. *arXiv*. 90

Tapaswi, M., Zhu, Y., Stiefelhagen, R., Torralba, A., Urtasun, R., and Fidler, S. (2015). Movieqa: Understanding stories in movies through question-answering. *arXiv*. 90

Technologies, V. (2022). Veo sports camera. `https://www.veo.co/en-us`. Accessed: 2022-09-01. 83

ThePhD (2018). sol2: a c++ <-> lua api wrapper (version 2.20.6). [Software]. https://github.com/ThePhD/sol2. 113

Tomlinson, B., Blumberg, B., and Nain, D. (2000). Expressive autonomous cinematography for interactive virtual environments. In *Proceedings of the fourth international conference on Autonomous agents - AGENTS '00*. ACM Press. 68

Turner, R., Balaguer, F., Gobbetti, E., and Thalmann, D. (1991). Physically-based interactive camera motion control using 3d input devices. In *Scientific Visualization of Physical Phenomena*, pages 135–145. Springer Japan. 68

Unity (2021). Unity (version 2021.3.3f1). [Software]. https://www.unity.com. 54, 65

Unity (2022). Cinemachine (version 2.8.4). [Software]. https://unity.com/unity/features/editor/art-and-design/cinemachine. 42, 62, 68, 88

Vázquezz, P.-P., Feixasz, M., Sbertz, M., and Heidrich, W. (2001). Viewpoint selection using viewpoint entropy. In *VMV*, pages 273–280. Citeseer. 31, 71

Vicol, P., Tapaswi, M., Castrejon, L., and Fidler, S. (2018). MovieGraphs: Towards understanding human-centric situations from videos. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE. 90

Ware, C. and Osborne, S. (1990). Exploration and virtual camera control in virtual three dimensional environments. In *Proceedings of the 1990 symposium on Interactive 3D graphics - SI3D '90*. ACM Press. 36, 37, 51, 60

Wei, S.-E., Ramakrishna, V., Kanade, T., and Sheikh, Y. (2016). Convolutional pose machines. In *CVPR*. 164

Weiss, T., Litteneker, A., Duncan, N., Nakada, M., Jiang, C., Yu, L., and Terzopoulos, D. (2018). Fast and scalable position-based layout synthesis. *IEEE Transactions on Visualization and Computer Graphics*, pages 1–1. 72, 86

Weiss, T., Litteneker, A., Jiang, C., and Terzopoulos, D. (2017). Position-based multi-agent dynamics for real-time crowd simulation. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, page 27. ACM, ACM. 72

Wengert, R. E. (1964). A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464. 140, 151

Wu, H.-Y., Galvane, Q., Lino, C., and Christie, M. (2017). Analyzing elements of style in annotated film clips. *Eurographics Workshop on Intelligent Cinematography and Editing*. 90

Wu, H.-Y., Palù, F., Ranon, R., and Christie, M. (2018). Thinking like a director: Film editing patterns for virtual cinematographic storytelling. *ACM Transactions on Multimedia Computing, Communications and Applications*, 14(4):1–23. 85

Xiao, D. and Hubbold, R. (1998). Navigation guided by artificial force fields. In *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '98*. ACM Press. 75

Yoo, J. E., Seo, K., Park, S., Kim, J., Lee, D., and Noh, J. (2021). Virtual camera layout generation using a reference video. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM. 75

Young, R. C. (1931). The algebra of many-valued quantities. *Mathematische Annalen*, 104(1):260–290. 125

Yu, L.-F., Yeung, S.-K., Tang, C.-K., Terzopoulos, D., Chan, T. F., and Osher, S. (2011). Make it home: automatic optimization of furniture arrangement. In *ACM SIGGRAPH 2011 papers on - SIGGRAPH '11*. ACM Press. 86

Zeleznik, R. and Forsbergt, A. (1999). Unicam - 2d gestural camera controls for 3d environments. In *Proceedings of the 1999 symposium on Interactive 3D graphics - SI3D '99*. ACM Press. 37