Virtual Camera Selection using a Semiring Constraint Satisfaction Approach

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Michael Robert Janzen

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

Players and viewers of three-dimensional computer generated games and worlds view renderings from the viewpoint of a virtual camera. As such, determining a good view of the scene is important to present a good game or three-dimensional world. Previous research has developed technologies to find good positions for the virtual camera, but little work has been done to automatically select between multiple virtual cameras, similar to a human director at a sporting event. This thesis describes a software tool to select among camera feeds from multiple virtual cameras in a virtual environment using semiring-based constraint satisfaction techniques (SCSP), a soft constraint approach. The system encodes a designer's preferences, and selects the best camera feed even in over-constrained or under-constrained environments. The system functions in real time for dynamic scenes using only current information (i.e. no prediction). To reduce the camera selection time the SCSP evaluation can be cached and converted to native code. This SCSP approach is implemented in two virtual environments: a virtual hockey game using a spectator viewpoint, and a virtual 3D maze game using a third person perspective. Comparisons against hard constraints are made using constraint satisfaction problems.

Acknowledgements

I would like to thank a number of people and organizations for their assistance in completing this thesis.

Thank you to my supervisors Mike Horsch and Eric Neufeld for their guidance, patience, and occasional prodding. My thesis topic bridges two areas in computer science. Both supervisors have lent me their expertise and encouraged me to further my studies in their respective areas. I also appreciate their guidance in publishing papers related to this thesis, and giving me the opportunity to present the work at conferences.

Thank you to my committee members for their guidance and direction: Ian McQuillan, Kevin Stanley, Tony Kusalik, and Longhai Li. Thank you to my external examiner, Scott Goodwin.

I would like to thank people in the IMG lab for stimulating conversations, good reading group discussions, and expanding my viewpoint of computer science beyond my research focus. Thanks also to our summer student, Stephen Damm, for his help improving the look of the simulation programs.

A graduate student does not work optimally on an empty stomach. I am very thankful for funding from NSERC, the Department of Computer Science, and my supervisors. Thank you also to my current employer, The King's University College, who took a risk and hired me prior to completing my PhD and provided time to complete later drafts.

I also appreciate the opportunity to teach during my studies. In addition to enjoying teaching, I found explaining concepts to undergraduate student useful practice for conference presentations, and distilling ideas in my own mind.

I would like to thank the technical and support staff in the Department of Computer Science, in particular Jan Thompson, who was always helping graduate students stay on top of their administrative responsibilities, as well as time for a smile and chat to see how students are doing. Also thank you to Heather Webb for helping to complete various forms and other paperwork; completing travel expense forms still seem a bit of a magical process to me.

Lastly, I would like to thank my family. I know they are immensely proud of me even though I sometimes suspect they only pretend to understand what I am explaining to them about my thesis. Their thoughts and prayers are always an encouragement. Thank you to my wife Alisha for love, support, encouragement, and making the move to Saskatchewan a great adventure for the two of us.

Contents

List of Tables

List of Figures

List of Abbreviations

AC	arc-consistency
CSP	constraint satisfaction problem
DCCL	Declarative Camera Control Language
EAC	existential arc-consistency
EDAC	existential directed arc-consistency
FAC	full arc-consistency
FDAC	full directed arc-consistency
NC	node consistent
NDC	normalized device coordinates
PCSP	partial constraint satisfaction problem
PVR	Potential Visibility Region
SCSP	semiring constraint satisfaction problem (also semiring-based constraint satisfaction problem)
VAC	virtual arc consistency
VCSP	valued constraint satisfaction problem
WCSP	weighted constraint satisfaction problem

Chapter 1

Introduction

When filming a real television show, movie, or sports event, various camera persons control individual cameras, and a director selects which of the camera feeds will be displayed on the screen. This camera selection involves a trade-off, as different cameras display different subjects, objects, angles, and zoom levels. A corresponding virtual director is needed in the virtual domain to select between available virtual cameras. To manage the trade-off, the virtual director, as presented in this thesis, uses a soft constraint satisfaction approach to balance different visual aspects of the scene.

Some animation is scripted, such as movies. Other animation is primarily unscripted, such as video games. In a scripted scene, the director can arrange for a coordination between the selected camera and action in the scene. In an unscripted scene, however, camera selection must react to the scene since action cannot be predicted. For this reason, different techniques are used in scripted and unscripted scenes.

In this document, a scene with unpredictable action will be called a *dynamic scene*. A scripted scene is entirely predictable; all future states are known when the scene begins. A scene in which all objects and characters are placed at random at any given point in time is the most dynamic scene possible, in that it is the least predictable. Action in other scenes falls in between these two extremes, in that action is coherent, but not entirely predicatable.

The main prior work on camera selection is He *et al.*'s idiom approach [49] described more fully in Chapter 3. While suitable for scripted scenes, it is unsuitable for dynamic scenes. In scripted scenes only a small number of good transitions between cameras need be specified, reflecting a director's style and the rules of cinematography [64, 3]. The number of possible transitions is usually low; when constructing idioms, Christenson *et al.*'s example considers selection from at most two possible cameras at a time [18]. Guards are conditions that must be satisfied before the system can transition to a different state; for example, a maximum duration of time. Thus, when a guard on a transition is satisfied, the finite state machine selects the next camera feed.

In highly dynamic scenes, the best camera feed could come from any camera at any given moment. When guards on multiple transitions are satisfied, the finite state machine is unable to determine the best transition to select, rather, all transitions outgoing from the current state are

considered equally preferable. A similar problem occurs if no guard is completely satisfied, the current feed is displayed even if another camera feed is more preferable. The problem of camera selection in dynamic scenes is currently unsolved - referring to He's Virtual Cinematographer and through the lens systems, Turkay *et al.* [82] mention, "All of these techniques require expert users or predefined constraints and [are] not suitable for dynamic and crowded scenes".

Previous works have focused on camera placement, which selects a location and angle for a virtual camera in a virtual world (see Chapter 3). These systems could be used to place cameras for the camera selection system to select among. Some of these systems use a constraint satisfaction problem (CSP) approach (see Chapter 2.1.1 for the background on CSPs). Camera placement strategies based on classical constraints (see Chapter 3.2), like camera selection methods, are unable to rank solutions when there is more than one solution, and fail when there is no solution that satisfies all the constraints. An exception is Bourne *et al.*'s weighted CSP (WCSP) approach. Under a WCSP interpretation, a constraint is either satisfied or unsatisfied; its effect on the overall selection is governed by the weight assigned to the given constraint. Constraints, however, cannot be partially satisfied; gross violations of the constraint cost the same as slight violations. Also, Bourne's solution does not satisfactorily address the problem of camera selection, but rather he states that a method better than He *et al.*'s solution is required [16].

Given infinite memory and processing power, the problems of virtual camera placement and virtual camera selection become equivalent. Selecting a camera feed from an infinite number of possible camera positions and orientations is equivalent to placing the camera in the scene. Alternatively, a camera that is able to move to an arbitrary position and orientation in the scene can produce the same output as a camera selection algorithm. Given limited resources, however, solutions to the camera placement problem and the camera selection problem take different approaches. For example, camera placement algorithms can restrict their search to nearby positions, while camera selection algorithms can consider a limited number of cameras that have sufficiently different views from one another.

Camera selection can be solved as an optimization task, since one camera feed is displayed on a display at any given time. That is, given the context, the camera that maximizes the designer's preferences should be chosen. A *designer* is a human who specifies preferences, whereas a *director* is a human or virtual entity that selects a camera based on the designer's preferences. As in the camera placement problem described above, previous camera selection work treats camera selection as a satisfaction task. The main problem with this approach is that solutions are ranked using a binary ranking: the set of constraints is either satisfied or unsatisfied. Consequently, suboptimal camera selection may result when multiple solutions satisfy all constraints, or when no solution fully satisfies all of the constraints.

As its main contribution, this thesis presents a method to select camera feeds from among

a set of camera feeds in dynamic virtual scenes by modularly encoding a designer's preference in independent functions (see the design in Chapter 5). These preferences combine under the semiring-based constraint satisfaction techniques framework (SCSP) constructing a global preference yielding a total order. Presented in Chapter 7, when given a designer's preferences the SCSP solver can select, in real time, the optimal camera feed from up to thousands of camera feeds. The trade-off of number of constraints, camera feeds, and displays is also presented in Chapter 7. Modularly encoding preferences is valuable since it provides the possibility of code reuse. Additionally, it may be less fragile than a hand coded system (a rule-based system for example), since preferences are combined in a systematic fashion.

The designer's preferences are represented in static constraints, which remain fixed for a particular SCSP solution, while dynamic constraints change to reflect current circumstances in a dynamic scene. Thus, the dynamic constraints are set by the system at run time to align solutions with the current circumstances. The tuple with the highest preference in the total order indicates which cameras to select for one or more displays. The system expands to automatically handle multiple displays, providing a convenient method for increasing complexity in the SCSP system. Additional displays require the algorithm solving the SCSP to optimize for multiple output variables. The system is demonstrated in two domains: a spectator hockey game, and a third person perspective maze game, presented in Chapters 4.1 and 4.2 respectively. These domains use OpenGL to render the graphics, which is introduced in Section 2.3.

As the number of constraints increase, the time required to find the optimal solution increases. To decrease the average time required to find a solution, the select camera feeds from previous SCSP solutions are cached. If the state of the scene, encoded by the dynamic constraints, reoccurs, the set of cameras to select is retrieved from the cache rather than performing another search. The cache can also be converted to native code to decrease the time needed to select camera feeds. These acceleration techniques are explained in Chapter 7.

Chapter 8 presents results in which different camera selection profiles result in different camera selection. Section 8.3 compares the camera feeds selected using a SCSP method with a classical CSP implementation. Conclusions and future work are presented in Chapter 9.

Chapter 2

Background

This thesis work combines areas of artificial intelligence with computer graphics. Mainly, it uses artificial intelligence techniques to solve a previously unsolved problem in computer graphic animation. The subfield of artificial intelligence used is CSPs and its generalization to soft constraints, discussed in Sections 2.1.1 through 2.1.4. Using OpenGL for generating real time graphics, a simulator is presented to experimentally apply the solution. The background to introduce the OpenGL system is also discussed in this chapter (Section 2.3).

2.1 Constraint Satisfaction Problems

2.1.1 Classical Constraint Satisfaction Problems

A CSP is defined as a tuple $\langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$ where $\mathbf{X} = \{X_1, \ldots, X_n\}$ is a set of variables, $\mathbf{D} = \{D_1, \ldots, D_n\}$ is a set of domains, one for each variable, and $\mathbf{C}$ is a set of constraints between variables [66]. For example, a binary constraint, $C_{ij} \subset D_i \times D_j$, specifies allowed combinations of the domain values for the variables X_i and X_j. This section will consider CSPs with only binary constraints, since a CSP with n-ary constraints can be converted to an equivalent CSP where all constraints are either unary or binary [28]. This thesis considers domains of $\mathbf{D}$ that are finite, i.e., $D_i = \{x_{i,1}, \ldots, x_{i,k_i}\}$. A solution to a CSP is an assignment, $(X_1 = x_{1,i_1}, \ldots, X_n = x_{n,i_n})$, such that all constraints in $\mathbf{C}$ are satisfied, i.e., for each $C_{ij} \in \mathbf{C}$, $\exists (x_{i,u}, x_{j,v}) \in C_{ij}$. For convenience this assignment is written as $(X_1 = x_1, \ldots, X_n = x_n)$, where each value assigned to X_i is from its corresponding domain, D_i, as done in Dechter's book [30].

There are several search methods used to find a solution to a CSP. Generate-and-test is a naive method that enumerates a sequence of assignments until a solution is found, or all possibilities have been considered [55]. A technique called *backtracking search* explores the search space of possible assignments in a depth-first fashion. Backtracking search can be considered as an implementation of generate-and-test, but enables optimizations that a naive generate-and-test algorithm would not employ. In backtracking search, the search space is organized in a tree structure where each node, N_i, corresponds to a variable, X_i, and the branches out of N_i correspond to the values from D_i. Thus each internal N_i (nodes with both branches in and branches out) corresponds to a

partial assignment because some variables are assigned values while others are not. For example, if variables are considered in a left to right ordering, then a partial assignment is $(X_1 = x_1, \ldots, X_{i-1} = x_{i-1}, X_i, \ldots X_n)$ for $1 < i \leq n$. At the root node no values are assigned to variables and a leaf node represents an assignment (also called a complete assignment). In a consistent partial assignment, for each combination of variables assigned values, $X_i = x_i$ and $X_j = x_j$, $(x_i, x_j) \in C_{ij}$. Conversely, in an inconsistent partial assignment there is at least one pair such that $(x_i, x_j) \notin C_{ij}$. A consistent complete assignment is a solution.

Backtracking search traverses a tree structure of partial assignments, starting at the root node, until it reaches a leaf node or encounters an inconsistent partial assignment. Upon reaching a leaf node that corresponds to a solution, the search may be terminated and the solution returned. Alternatively, the search can continue exhaustively in order to return all solutions. Upon encountering an inconsistent assignment or inconsistent partial assignment, the search backtracks to a node with a consistent partial assignment (or root node) and continues along one of its unexplored branches. In this way, sections of the search space with an inconsistent partial assignment are *pruned*, in that they are not explored, since further exploration of such spaces cannot lead to a solution. If the search traverses all branches in the tree without finding a solution then the search terminates and returns the empty solution.

In a backtracking search, the same consistency violation may occur in different branches of the tree search, a problem called *thrashing* [63]. For example, consider a CSP where variables are assigned values by backtracking search in the order X_1, X_2, and X_3. Suppose X_1 is assigned x_1 for which there is no consistent value in D_3. After assigning $X_1 = x_1$, $X_2 = x_2$, and $X_3 = x_3$, an inconsistency between X_1 and X_3 is detected, i.e. $(x_1, x_3) \notin C_{13}$, causing the backtracking algorithm to change the value at X_2, which does not remove the inconsistency between X_1 and X_3.

A solution to thrashing is *domain revision* which removes inconsistent values from the domains of variables [63]. Continuing with the above example, since there is no value in D_3 that is consistent with $X_1 = x_1$ then the value x_1 could be removed from D_1. This is more formally expressed as $\neg\exists x_j \in D_3$ such that $(x_1, x_j) \in C_{13}$. The removal results in another CSP where solutions to the new CSP are also solutions to the original CSP. Additionally, because only inconsistent values are removed, a solution to the CSP before domain revision is also a solution to the revised CSP.

A simple application of domain revision is *forward checking* [48]. With forward checking when X_i is assigned a value, x_i, the domain of every unassigned variable X_j that shares a constraint with X_i is revised such that $(x_i, x_j) \in C_{ij}, \forall x_j \in D'_j, D'_j \subseteq D_j$, where D'_j is the revised domain of D_j. In the above example, if there is no consistent value for X_3 when assigning $X_1 = x_1$, the domain D_3 will become empty. When D_3 becomes empty the algorithm prunes the branch $X_1 = x_1$, and avoids the thrashing in the above example.

Another application of domain revision is *arc-consistency* [63]. An arc is a constraint between

two variables[1]. A constraint, C_{ij}, is arc-consistent if $\forall x_i \in D_i \; \exists x_j \in D_j$ such that $(x_i, x_j) \in C_{ij}$. Inconsistent arcs can be made consistent by removing values from D_i that cause the inconsistency. A CSP is arc-consistent if all of its arcs are arc-consistent. Note that an arc-consistent CSP of the above example would not experience the thrashing in the example, as x_1 would be removed from D_1 making C_{13} arc-consistent.

Algorithms for ensuring arc-consistency can return an arc-consistent CSP given an input CSP by removing domain values until consistency is reached [63]. The algorithm AC-1 iterates over the constraints, removing values at each iteration until an iteration completes without removing any domain values [63]. Another algorithm, AC-3, uses a queue of constraints to reduce the number of times a constraint is examined [63]. (Following Macworth's naming, AC-3 rather than AC-2 is used [63].) Initially all constraints are put on the queue. Each constraint is removed from the queue for examination and the algorithm terminates when the queue is empty. Similar to AC-1, constraints are made consistent by removing values from domains. If an element is removed from the domain of D_i then all of the constraints associated with X_i are put on the queue unless the constraint is already on the queue. Note that if C_{ij} causes the constraints of X_j to be put on the queue, it is unnecessary to put C_{ji} on the queue. Other algorithms, such as AC-4, AC-6, AC-7, AC-8, AC-2000, AC-2001, etc., balance space with time and can more closely match worst case complexity with the theoretical limit [73]. If e is the number of arcs and d is the size of the largest domain, the time complexity of AC-3 is $O(ek^3)$[30]. AC-4 achieves a time complexity of $O(ek^2)$ by storing the amount of support for each domain from another variable's domain [30].

Arc-consistency is sound, but not complete. In some cases of domain revision, the size of each domain may become one, yielding a solution directly. Other times domain revision may result in an empty domain. If the domain revision occurs before starting the search process, then an empty domain results in the empty solution to the CSP (i.e. no solution). If the domain revision occurs during a search process, then an empty domain causes the search to backtrack. When there are multiple solutions to a CSP, enforcing arc-consistency can yield another CSP with fewer branches in its backtracking search tree.

An alternative to domain revision for solving the thrashing problem is *backjumping*. When a backjumping algorithm encounters a constraint violation it attempts to jump as early as possible in the search tree without missing solutions. Continuing with the X_1, X_2, and X_3 example, Gaschnig's backjumping algorithm would return to the X_1 node upon a conflict detected at X_3 since X_1 is the lowest variable in the search tree in conflict with X_3 [42]. A graph-based backjumping algorithm would also return to X_1 since the graph structure indicates that X_1 is the earliest variable that could be involved in the constraint violation [40, 29]. As an improvment to Gasching's backjumping,

[1] Constraints are called arcs when a CSP is represented graphically. Here the term arc-consistency is used, rather than constraint-consistency, to align with published literature.

graph-based backjumping methods are able to jump at internal dead nodes as well as at leaf nodes. A conflict-directed backjumping algorithm combines the two backjumping algorithms to potentially jump further than either method alone [72].

2.1.2 Partial Constraint Satisfaction

Some real world applications result in over-constrained CSPs in that they have no solution. In these cases the solution to the problem can be redefined as an assignment that violates the fewest number of constraints, a variant of CSP termed MaxCSP [41]. A user, however, may want to specify that some constraints are more important to satisfy than others.

A constraint hierarchy specifies hard constraints and soft constraints [14, 15]. Hard constraints must be satisfied to consider an assignment a solution. It is preferable that soft constraints are satisfied, but an assignment containing unsatisfied soft constraints will still be considered a solution. Two methods for solving constraint hierarchies are refinining algorithms and local propagation techniques. Refining algorithms prioritize satisfying hard constraints followed by attempting to satisfy the most preferred soft constraints [86, 60]. Local propagation techniques determine a value for a variable and then propagate that assignment to other constraints (the value for a variable may be set by a user) [74, 60]. These local propagation techniques can differ from arc consistency. An example is DeltaBlue, which is an incremental algorithm [74]. DeltaBlue selects a value for a variable based on a constraint, and propogates the effects to other variables. The selection of this value changes the problem, where arc consistency algorithms return an equivalent problem. The constraint hierarchy directs which soft constraints may be changed when satisfying hard constraints.

A partial constraint satisfaction problem (PCSP) is a generalization of MaxCSP where solutions are assignments that violate an acceptable number of constraints or an acceptable degree of consistency (cost, degrees of preference, etc.) [41, 12]. A valuation is the measure of the degree of consistency. Another variation of a PCSP is a WCSP, where each constraint is assigned a weight [41, 79]. The valuation is the sum of the weights of the consistent constraints. This section will consider PCSPs with regards to the number of violated constraints, but WCSPs or other measures of consistency follow in a straight-forward fashion.

A search algorithm for solving a PCSP, called *branch and bound*[2], is similar to backtracking search presented in Chapter 2.1.1. Branches are pruned when the number of constraint violations of the current partial assignment exceeds a threshold, called the lower bound, rather than pruning when encountering the first violation [56]. The current best solution is a good threshold, and its use is almost universal. To improve the performance of branch and bound, search nodes may be considered in a best first order, based on variable and value selection heuristics [30]. Rather than traverse the tree in a left to right fashion, partial assignments with fewer violations are considered

[2]Branch and bound is widely used for optimization tasks, not only for solving PCSPs.

before partial assignments with more violations. This approach may evaluate fewer nodes, but its memory use can approach breadth first search for large problems.

Many of the considerations involved with searching in CSPs have analogies for MaxCSP and PCSP, including consistency propagation [41]. For example, Cooper and Schiex generalized the arc consistency property from classical CSPs to WCSPs [23, 26]. Propagation techniques with soft constraints are discussed more fully in Section 2.2.

2.1.3 Valued Constraint Satisfaction Problems

A valued constraint satisfaction problem (VCSP) is a tuple $(\mathbf{X}, \mathbf{D}, \mathbf{C}, \mathbf{\Sigma})$ [76, 75]. As before $\mathbf{X}$ is a set of variables, and $\mathbf{D}$ is a set of corresponding domains. $\mathbf{C}$ is a set of cost functions, similar to preferences in a SCSP, and $\mathbf{\Sigma} = (\mathbf{E}, \oplus, \succcurlyeq)$. $\mathbf{E}$ is a set of *valuations* totally ordered by $\succcurlyeq$ and combined using $\oplus$, which is closed. As examples, a valuation can encode a cost, weight, or penalty. A SCSP generalizes a VCSP; if there is a total ordering on the elements of $\mathbf{A}$ in the SCSP (similar to $\mathbf{E}$ in a VCSP), then the SCSP can be represented using a VCSP [23].

As with classical CSPs, PCSPs, and SCSPs; VCSPs can be made arc-consistent [23, 25]. When the VCSP is binary, then general arc-consistency is also known as soft arc-consistency [26] (Section 2.2).

2.1.4 Semiring Constraint Satisfaction Problems

A SCSP is an abstract generalization for CSP and CSP-like problems. The abstraction identifies common aspects of these problems and allows variants to be specified in a general way. For example, classical CSPs and PCSPs are both instances of a SCSP [10]. Another variation, called Fuzzy CSPs, will be explained later in this section after introducing a SCSP more formally [12].

The key idea in the generalization of SCSPs is to represent constraints as functions rather than relations, as is the case with CSPs. In order to generalize different CSP-like variations, the SCSP requires that the function return values that can be combined and compared in a few restricted ways. The theory of SCSPs indicates what properties are essential to CSP-like problems and allows a wide variety within these properties.

Formally, a SCSP is a tuple $(S, \mathbf{X}, \mathbf{C}, \mathbf{D})$ where S is a semiring $\langle \mathbf{A}, +, \times, \mathbf{0}, \mathbf{1} \rangle$, $\mathbf{C}$ is a set of constraints as explained below, and $\mathbf{X}$ and $\mathbf{D}$ are variables and domains as defined in Section 2.1.1. The domains of the variables are typically discrete and finite [11, 12]. In a semiring, $\mathbf{A}$ is a set containing $\mathbf{0}$ and $\mathbf{1}$. $\mathbf{A}$ represents the set of preference levels, or consistency. The symbols $\mathbf{0}$ and $\mathbf{1}$ depict the minimal and maximal values in the set $\mathbf{A}$[3]. The operators $+$ and $\times$ operate over pairs of elements of $\mathbf{A}$ and are closed and associative. For use in constraint propagation, it is necessary to

[3]Bold $\mathbf{0}$ and bold $\mathbf{1}$ are symbols and not the numeric values 0 and 1

impose the additional constraint that + be idempotent, i.e., $a+a=a$, in which case the semiring is called a c-semiring [12]. For the operator $\times$, **1** acts as the unit element and **0** acts as the absorbing element [4]. For the operator +, **0** acts as the unit element [12, 9].

As an example, a classical CSP can be implemented in a SCSP framework using this notation. $\mathbf{A} = \{false, true\}$, the operator + is implemented with $\vee$ (OR) and the operator $\times$ is implemented with $\wedge$ (AND) [12]. A Fuzzy CSP is defined as $\langle\{0,1\}, max, min, 0, 1\rangle$ where *max* and *min* return the maximum and minimum value of the pair of elements, respectively [36, 9, 24].

The set of constraints, **C**, is a set where each constraint is a function from variables included in the constraint to a preference value. Thus the constraint defines a function over some of the variables from **X**. Constraints are not necessarily binary. Constraints can be joined using the join operator, denoted $\otimes$, by using the $\times$ operator for each value in the domains of the variables included in the functions. An example is provided later in this section using *min* as the join operator and a global preference shown in Figure 2.1. Thus joining two constraints, C_i and C_j is denoted as $C_i \otimes C_j$. The combination of all constraints, using the $\otimes$ operator, yields the global preference over the problem. Specifying the global preference directly can be intractable in practice, since the number of tuples is exponential in the number of variables in **X**; specifying independent constraints in multiple, smaller tables is more tractable. Typically, for implementation, the global preference is obtained by a search, using the branch and bound method presented in Section 2.1.2, rather than constructing the global preference table in memory.

When preferences are specified such that there is a total order (i.e. all values are comparable) the join operation enables a ranking over the global preference [41]. A solution to a SCSP is a tuple that has the maximum global preference. Chapter 5 defines a semiring and preference functions such that a solution to the SCSP will indicate which camera should be chosen.

It may be that the solution found to a SCSP has an unacceptable preference level. In this case the SCSP could be relaxed to form a new SCSP, and the solution to the new SCSP may have an acceptable preference level [61, 60]. For example, a SCSP involving monetary profit could be adjusted to identify variables that limit the assignment of a high profit to the profit variable. A metric can be defined to identify how much each constraint differs from the corresponding original constraint [43, 60].

As an example implementation, SCSPs have been used to encode scheduling for steel making (hot strip mill) [60]. Related VCSPs, described in Section 2.1.3, have been used to encode the radio link frequency assignment problem (RLFAP) [17]. In the RLFAP, broadcasting towers are to be assigned frequencies such that they operate without noticable interference.

As a further example of a SCSP, consider a decision maker planning a meal consisting of a main dish, a fruit, and a drink. Suppose the decision maker considers the main dish to be independent

[4] The unit element is sometimes refered to as the identity element in other areas of math.

of the fruit and drink, but that the choice of drink is based on the fruit chosen. The preferences for this example can be specified in the two functions shown in Figure 2.1, showing that stir fry is preferred to pasta and an apple with water is the most preferred combination of fruit and drink. The example assumes that preferences lie between zero and one inclusively where one is the most preferred and zero is the least preferred.

Main Dish	Preference
pasta	.5
stir fry	.8

Fruit	Drink	Preference
apple	water	.9
apple	milk	.2
pear	water	.5
pear	milk	.1

Figure 2.1: Example Preferences

Obviously, the best meal is {stir fry, apple, water}. This is a simple example. The best choice is not always so obvious. The algorithm, however, must compute the preferences of tuples to determine the highest preference. To find the best combination of meal, constraints can be joined to construct the global preference. To this end the $\times$ operator combines preferences from the two tables' entries (and by extension any number of tables). Potential implementations of the $\times$ operator include the Fuzzy operator in which the combined preference is defined as the minimum of the input tuples [12]. Another implementation will be defined in Chapter 5. Thus, using the Fuzzy operator, each table entry of the joined function is defined as follows:

$$\begin{aligned} C_{join} &= C_i \otimes C_j(x_{i_1}, \ldots, x_{i_{k_i}}, x_{j_1}, \ldots, x_{j_{k_j}}) \\ &= C_i(x_{i_1}, \ldots, x_{i_{k_i}}) \times C_j(x_{j_1}, \ldots, x_{j_{k_j}}) \\ &= min(C_i(x_{i_1}, \ldots, x_{i_{k_i}}), C_j(x_{j_1}, \ldots, x_{j_{k_j}})) \end{aligned}$$

Note that if variable x_k appears in both C_i and C_j, then it appears only once in $C_i \otimes C_j$. The same variable may appear in C_i and C_j, so long as C_i and C_j return preference values to combine. Table 2.1 shows the global preference for the meal example using the Fuzzy operator for combining preferences where the best meal is stir fry, an apple, and water.

For large problems, storing the global preference as a table in memory can require an intractable amount of memory. Instead, the global preference can be examined one tuple at a time using search techniques described previously, such as branch and bound described in Section 2.1.2. This is similar to the classical CSP generate-and-test search, where all tuples of the global constraint could be generated, but backtracking search is used instead.

Also similar to classical CSP problems, constraint propagation can improve the performance of a search algorithm in SCSP problems. Similar to classical constraint propagation algorithms, such

Main Dish	Fruit	Drink	Preference
pasta	apple	water	$min(.9, .5) = .5$
pasta	apple	milk	$min(.2, .5) = .2$
pasta	pear	water	$min(.5, .5) = .5$
pasta	pear	milk	$min(.1, .5) = .1$
stir fry	apple	water	$min(.9, .8) = .8$
stir fry	apple	milk	$min(.2, .8) = .2$
stir fry	pear	water	$min(.5, .8) = .5$
stir fry	pear	milk	$min(.1, .8) = .1$

Table 2.1: Example of Combined Preferences using the Fuzzy combination operator

as AC-3 described in Section 2.1.1, SCSP propogation algorithms can reduce the search space to examine or prioritize searching more promising branches of the tree before less promising branches (such as xAC) [51]. These propogation algorithms are discussed in more detail in Section 2.2.

2.2 Constraint Propogation in Partial Constraint Satisfaction Problems

Similar to constraint propagation in CSPs, propagation techniques can be applied to soft constraints satisfaction problems to transform them into an equivalent problem. Equivalent means that a solution to the transformed problem is also a solution to the original problem. Typically, the transformed problem is simpler than the original problem (unless, of course, the original problem is already consistent). Many of these techniques hinge on the idea of improving the lower bound, as well as removing domain values. Since the lower bound in branch and bound search typically consists of costs, researchers have designed algorithms to project costs from binary constraints onto the lower bound. The technique was developed in stages and some of the algorithms in this process are presented in this section. The literature for a given technique typically refers to the algorithm that enforces a property and the property itself using the same name. The meaning is usually clear from the context of the sentence.

2.2.1 W-AC2001

As given, a constraint problem may be inconsistent. There may be a value in a domain that is never assigned to any variable in any solution. For example, when colouring a political map, a unary constraint may specify that no country can be coloured blue, since blue is reserved for colouring

water. In such an example, assigning blue to a country will lead to an inconsistency, so the colour blue could be removed from the domain of colours to assign to countries. Schiex *et. al* use the idea of consistency to transform a soft constraint problem using their AC2001 algorithm [75]. Scheix *et. al*'s algorithm is designed to work with MAX-CSP problems, in which a solution satisfies the maximum number of constraints. Larrosa adapts and extends their definition of consistency, used in the W-AC*2001 algorithm for use with WCSPs [57].

Two key consistency measures in both AC2001 and W-AC*2001 are node consistency and arc-consistency, which consider binary and unary constraints. From W-AC*2001, based in the framework of a VCSP, the implementation is based on additive cost, where the cost sums when joining tuples.

$$a \oplus b \quad = \quad \min\{\top, \text{a} + \text{b}\}$$

where $\oplus$ combines values from $\mathbf{E}$, and $\top$ is the maximum value in $\mathbf{E}$. A solution is a complete assignment, with the lowest cost.

A value is said to be *node consistent* if its valuation is less than the maximum cost, $\top$. A variable is node consistent if all of its values are node consistent. A value to be assigned to a variable i in a binary constraint, is arc-consistent with respect to a given constraint C_{ij} if it is node consistent and there is support from C_{ij}. To have support from constraint C_{ij} means that there is an assignment such that the resulting tuple has the minimum cost, denoted $\bot$.

Variables can be made node consistent by removing inconsistent values from their domains. Constraints can be made arc-consistent by projecting some costs onto unary constraints. When a cost from a binary constraint, C_{ij}, is projected onto a unary constraint, C_i, the valuation from the C_{ij} is added to C_i. The lowest cost from C_{ij} for a given value of $X_i = x_i$ is subtracted from the valuation of all tuples in C_{ij} where $X_i = x_i$. To handle tuples with a valuation of $\top$, anything subtracted from $\top$ results in $\top$, since the join $a \oplus b$ is the minimum of $\top$ or $a + b$. A WCSP is arc-consistent if all of its constraints are arc-consistent.

To help clarify the above, consider an example from [57] for W-AC2001, the adaptation of the AC2001 algorithm for weighted constraints. Here the notation from Larossa's paper is adapted to be more similar to notation in this thesis. In the example $\mathbf{X} = \{x, y\}$, $x \in D_x$, $D_x = \{a, b, c\}$, $y \in D_y$, $D_y = \{a, b, c\}$, $\mathbf{E} \in \{0, 1, 2, 3\}$, $\bot = 0$, and $\top = 3$. There are two unary constraints, shown in Tables 2.2 and 2.3, and one binary constraint shown in Table 2.4.

A solution is $x = b$, $y = b$ since there is no assignment with a lower additive cost. The example is initially not node consistent, since the assignment $x = c$ results in the tuple being assigned the maximum cost of 3. Therefore c should be removed from D_x. The WCSP is not arc-consistent since, when x is assigned a, there is no assignment to y in the binary constraint that has a cost of

x	Cost
a	0
b	1
c	3

Table 2.2: Example Unary Constraint over x

y	Cost
a	2
b	1
c	2

Table 2.3: Example Unary Constraint over y

$\bot$. The cost of 1, from the tuple $\{x = a, y = c\}$, can be projected onto $y = c$. The tuple from the unary constraint with $y = c$ increases by 1 to 3, and all tuples in the binary constraint with $y = c$ decrease by one. Now y is no longer node consistent, however, since $y = c$ is assigned a value of $\top = 3$. Therefore c should be removed from D_y. The results are shown in Tables 2.5, 2.6, and 2.7:

After the removal of c from D_y, the WCSP is not arc-consistent since there is no assignment to $x = a$ with a cost of $\bot$ for the binary constraint. The cost from $x = a, y = a$ can be projected onto $x = a$, resulting in a cost of 1 for the tuple with value a in the unary constraint over x, and a decrease by 1 in all tuples from the binary constraint where $x = a$. The WCSP is now node and arc-consistent.

Notice that now the unary constraint over x has a cost of 1 whether $x = a$ or $x = b$. Larrosa extends the idea of node consistency in the algorithm W-AC*2001 to include this cost in a new unary constraint, he calls C_0. The value 1 can be subtracted from all tuples in the unary constraint over x, and added to C_0. When determining node consistency, C_0 is added to the valuation of the tuple. This version of node consistency is named NC*. Consequently, $y = a$ in the unary constraint over y now has a cost of $3 = \top$, so a can be removed from D_y. Thus, W-AC*2001 is able to remove more values and make the search space smaller than W-AC2001, but at a cost of increased computation.

2.2.2 Existential Directed Arc-Consistency

Section 2.2.1 introduced NC*, where the cost C_0 is added to variables' values' costs when determining node consistency. Similarly, when a value, b, is considered in support of a binary constraint,

x	y	Cost
a	a	1
a	b	2
a	c	1
b	a	0
b	b	0
b	c	2
c	a	0
c	b	0
c	c	0

Table 2.4: Example Binary Constraint

x	Cost
a	0
b	1

Table 2.5: Revised Unary Constraint over x after removing $x = c$

$C_{ij}(a, b)$, the unary cost from b can be added to the cost of the binary constraint. If this sum is equal to $\perp$, a constraint is considered full arc-consistent (FAC, FAC* if it also considered NC*) [27]. A variable is considered FAC* if all of its values have at least one FAC* binary constraint. A WCSP is considered FAC* if all of its variables are FAC*. The effect of using FAC* is the possible removal of more values from domains in the WCSP (i.e. additional prunning).

There is no guarantee, however, that a WCSP can be converted into a FAC* WCSP. Costs may transfer back and forth between variables, with no version of the WCSP having a FAC* property. If an order is applied to the variables, then a WCSP can be made full directed arc-consistent (FDAC*) [59]. In terms of the domains pruned, FDAC* prunes no more than FAC*.

y	Cost
a	2
b	1

Table 2.6: Revised Unary Constraint over y after projecting $\{x = a, y = c\}$ onto $\{y = c\}$ and removing removing $y = c$ from unary constraint over y

x	y	Cost
a	a	1
a	b	2
b	a	0
b	b	0

Table 2.7: Revised Binary Constraint after removing $x = c$ from D_x, projecting $\{x = a, y = c\}$ onto $\{y = c\}$, and removing $y = c$ from D_y

A binary constraint has simple support if it has a tuple with a cost of $\perp$. A binary constraint has full support if it has a tuple with the cost of $\perp$ and the unary constraint of the supporting variable also has a cost of $\perp$. A variable is existential arc-consistent (EAC*) if at least one value has a unary cost of $\perp$ and it has full support in each of its binary constraints [27]. de Givrey et al. show that this can cause a variable to cease being NC*; the cost from that variable can be absorbed into C_0, raising the lower bound on the problem. EAC* has stronger consistency than FDAC*, but weaker than FAC*. Existential directional arc consistency (EDAC*) incorporates both FDAC* and EAC*.

2.2.3 Virtual Arc-Consistency

Virtual arc-consistency (VAC) uses CSP techniques to raise the lower bound C_0 of a WCSP[5] [22, 26]. The WCSP uses non-negative integer weights. VAC can raise the lower bound, even when the WCSP is already EDAC*. Thus, it may be possible to prune more domain values from variables, as they may exceed $\top$ after the WCSP has been made VAC. Recall, as also used in the literature, the acronym VAC can refer to the algorithm or the level of consistency the algorithm enforces.

A key step in the VAC technique is to create a corresponding CSP, named Bool(P), from the WCSP named P. If a weight on a constraint is not equal to $\perp$ then it is false in Bool(P), otherwise the constraint is true. A problem P has the property VAC if Bool(P) is consistent. Cooper *et al.* show that if P is not VAC then the lower bound, C_0, can be raised until P is VAC.

The technique works in three phases. The first phase of the technique makes Bool(P) consistent using classical arc-consistency [22]. During the propagation, the removal of supports from other variables are recorded (i.e. for $C_{ij}(a, b)$, if $a = 1$, and $b = 2$ is found to satisfy the constraint then $b = 2$ is a support for $a = 1$). If P is not VAC, then there will be a domain wipeout for some variable of Bool(P), meaning that there is no consistent value to assign. This implies that C_0 could be raised by some amount. Currently, at this point in the algorithm, the amount is unknown and

[5] In [22] C_0 is named w_0.

is named λ.

The second phase of the algorithm uses the recorded information in reverse order to trace a minimal sequence of arc-consistency operations that result in the domain wipe-out. The trace can terminate when it reaches a value, call it d, with a cost greater than $\perp$ in the WCSP. This implies that λ can be no greater than d. Thus, the inferred value for λ is determined from the minimum non-zero cost that appears in the trace. If d is reached by more than one trace then the cost assigned λ from d is the cost of d divided by the number of traces that reach the value.

The third phase applies the sequence determined from phase one using soft consistency operations in the forward direction and the known value of λ, including the additional cost to C_0 from λ. Since the lower bound, C_0, has been raised by λ, additional pruning may be possible compared to the original WCSP. If λ may be fractional and P uses integer weights, λ is rounded up to the nearest integer.

The VAC algorithm may repeat, slightly increasing C_0 each time. A heuristic specifying the minimum increment for λ can then force a termination [26]. After the process of making P VAC, variables in P may not be NC*. Making P NC* can again raise the value of C_0, since the cost common to all values of a variable will be transfered into C_0.

2.2.4 xAC

Horsch *et al.* generalize the arc-consistency algorithm with an approach they term "xAC" [51]. The view underlying the algorithm is that arc-consistency approximates satisfiability, and that this approximation can be expressed in terms of the domains of single variables. Their approach is based in the SCSP framework, but does not depend on specific assumptions from the framework, such as the fact that $\mathbf{1}$ is an absorbing element for the $+$ operator. The projection of the solution to a single variable is called the *marginal*, a term borrowed from probability calculus. When the problem is tree structured, arc-consistency algorithms compute the marginal exactly. When the problem is not tree structured, arc-consistency algorithms approximate the marginal.

The marginal solution of a SCSP P on variable X can be determined from joining all constraints and projecting them onto the variable X:

$$M_X(P) \quad = \quad (\otimes_{c \in C} c) \Downarrow_{\{X\}}$$

Here C is the set of constraints. The operator $\Downarrow$ projects contraints onto X using the SCSP $+$ operator. The constraints are first joined using the $\times$ operator to combine tuples. Then tuples containing the same value for the X variable are compared using the $+$ operator. The mathematical operations for these operators depends on their definition in the semiring.

The computation of the marginal from the global solution is intractable in general. If the

problem is tree structured then the marginal on X can be determined from the marginals of the neighbours of X.

The algorithm iterates in steps until convergence, or a preset number of iterations is reached.

- The marginal distribution to each neighbour is computed (i.e. a variables value projected onto a neighbour through constraints that are over the two variables).
- Marginals are exchanged with all neighbours.
- Each variable updates its marginals using the information received from its neighbours.

In more detail, the neighbours' constraints are joined using the $\times$ operator, and projected onto X using the $+$ operator to combine tuples. If the problem is tree structured, then the marginal is exact after g iterations, where g is the diameter of the graph.

As an example, consider a fuzzy SCSP where the operators $+$ and $\times$ are instantiated using max and min, respectively. Suppose the SCSP contains the variables $\mathbf{X} = \{X, Y, Z\}$ with domains $\mathbf{D} = \{\{a, b\}, \{c, d\}, \{e, f\}\}$, and preferences ranging between 0 and 1. Three unary and two binary preferences are defined over the variables, as shown in Figure 2.2.

X	Pref
a	0.7
b	0.3

Y	Pref
c	0.9
d	0.8

Z	Pref
e	0.9
f	0.2

X	Y	Pref
a	c	0.3
a	d	0.7
b	c	0.5
b	d	0.8

Y	Z	Pref
c	e	0.3
c	f	0.9
d	e	0.4
d	f	0.7

Figure 2.2: Example preferences for demonstrating xAC algorithm

The problem is tree structured, has diameter $g = 2$, and Y separates X and Z. The global join is shown in Table 2.8. The marginals will converge after two iterations, shown in Figure 2.3. Notice that the projection of the global join will result in the same marginals shown in the second iteration in Figure 2.3.

X	Y	Z	Preference	X	Y	Z	Preference
a	c	e	0.3	b	c	e	0.3
a	c	f	0.2	b	c	f	0.2
a	d	e	0.4	b	d	e	0.3
a	d	f	0.2	b	d	f	0.2

Table 2.8: The global join in the xAC example

First Iteration:

X	Pref
a	0.7
b	0.3

Y	Pref
c	0.3
d	0.4

Z	Pref
e	0.4
f	0.2

Second Iteration:

X	Pref
a	0.4
b	0.3

Y	Pref
c	0.3
d	0.4

Z	Pref
e	0.4
f	0.2

Figure 2.3: Two example iterations of the xAC algorithm

When the problem is not tree structured, constraint propagation can continue until convergence or until a preset number of iterations. Marginals can be used in a branch and bound search as a value ordering heuristic.

2.3 OpenGL

The graphics in the hockey and maze games, discussed in Chapter 4, are rendered using OpenGL (Open Graphics Language). This section briefly introduces OpenGL, and the mathematics to transform polygons into pixels on the display. Additionally, some of the preference values of some constraints in the camera selection system depend on the location of objects projected to the display. These values are partially determined using information from the OpenGL pipeline, such as the location of the puck on a display in the hockey game, or the visibility of the player's avatar in the maze game.

OpenGL is based on the earlier Silicon Graphics' IRIS GL [77] with objects modeled as sequences of points [67][2][53]. Arbitrary polygons may be defined subject to the constraints that they are co-planar and convex[2]. Concave polygons are forbidden as they complicate shading algorithms and clipping algorithms, both described below.

2.3.1 Points and Transformations

OpenGL allows the definition of points in space and transformations to operate on those points, chiefly translations, rotations, and scaling operations which move, rotate, and adjust the size of polygons defined in the space. 3D points and 3D vectors are represented with four-valued vectors,

referred to as homogeneous coordinates [8]. In matrix form, homogeneous coordinates are as follows:

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

where x, y, and z are the values from three dimensions and w is zero for vectors and non-zero for points (typically $w = 1$) [2]. Note that, by convention, y is in the up direction, x is to the right, and z is out of the display towards the viewer.

The addition of the fourth dimension (w) enables both rotations and translations to be computed via matrix multiplication. For example, the matrix multiplications;

$$\begin{bmatrix} 1 & 0 & 0 & \alpha_x \\ 0 & 1 & 0 & \alpha_y \\ 0 & 0 & 1 & \alpha_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix},$$

perform a translation by a vector α and a rotation about the x axis by angle θ, respectively. Scaling by a vector β and shearing in the x direction can be performed with the following matrices:

$$\begin{bmatrix} \beta_x & 0 & 0 & 0 \\ 0 & \beta_y & 0 & 0 \\ 0 & 0 & \beta_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 1 & \cot(\theta) & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

Rotations about the y and z axes are performed with similar matrices, as are shears in the y or z direction[6]. Sequential transformations are multiplied into a single matrix by the associativity of matrices and are applied to points by a hardware implementation for increased speed [2].

To specify a rotation about an arbitrary vector it is useful to employ quaternions [50, 68, 80]. Quaternions enable smoother rotations than adjusting three angles about x, y and z axes. A quaternion is composed of a scalar plus a vector:

$$q = (s, (x, y, z))$$

A quaternion can be made into a unit quaternion by dividing through by its magnitude:

$$\hat{q} = \frac{q}{||q||}$$

[6] A shear can be constructed from a sequence of rotations, translations, and scalings but is important enough to consider it a basic transformation [2].

where

$$||q|| \quad = \quad \sqrt{s^2 + x^2 + y^2 + z^2}$$

A quaternion can represent a rotation about an arbitrary vector. A rotation of angle θ about a unit vector (x, y, z) is specified by the following:

$$Rot_{\theta,(x,y,z)} \quad = \quad [\cos(\theta/2), \sin(\theta/2) \cdot (x, y, z)]$$

To combine a rotation defined by a quaternion with matrices in the display pipeline, the quaternion must be converted to a rotation matrix [68]. A unit quaternion $(s, (x, y, z))$ corresponds to the rotation matrix:

$$\begin{bmatrix} 1 - 2 \cdot y^2 - 2 \cdot z^2 & 2 \cdot x \cdot y - 2 \cdot s \cdot z & 2 \cdot x \cdot z + 2 \cdot s \cdot y \\ 2 \cdot x \cdot y + 2 \cdot s \cdot z & 1 - 2 \cdot x^2 - 2 \cdot z^2 & 2 \cdot y \cdot z - 2 \cdot s \cdot x \\ 2 \cdot x \cdot z - 2 \cdot s \cdot y & 2 \cdot y \cdot z + 2 \cdot s \cdot x & 1 - 2 \cdot x^2 - 2 \cdot y^2 \end{bmatrix}.$$

2.3.2 Projections

In order to be viewed, a three-dimensional scene must be projected onto a two-dimensional display. Two such projections are orthographic projection (a.k.a. orthogonal projection), shown in Figure 2.4, and perspective projection, shown in Figure 2.5. An orthographic projection projects onto a plane by setting points' z coordinates to zero, assuming the viewer is looking down the z-axis, and the camera is at the origin with the up vector oriented along the positive y-axis.

The default view volume is a cube with corner points (-1,-1,1) and (1,1,-1). Polygons outside this volume are ignored during the projection. Polygons partially inside the volume are redefined as the portion enclosed by the volume, called clipping. Since clipping a convex polygon results in another convex polygon, clipping can be done simply (as opposed to clipping resulting in multiple polygons). If the view volume is not in its default orientation it must be transformed and the same transformation applied to all vertices in the 3D scene.

The perspective projection can be determined from simple Euclidian geometry. For a point at (P_x, P_y, $-P_z$) the projected value of P_x is $x^* = NP_x/(-P_z)$ and similarly $y^* = NP_y/(-P_z)$ where the viewer is located at (0,0,0) and N is the distance from the viewer to the view plane [53]. Rather than set z^* to the view plane distance, let $z^* = (aP_z + b)/(-P_z)$ where a and b are calculated using the far and near planes of the view volume.

$$a = -\frac{Far + Near}{Far - Near} \quad , \quad b = \frac{-2 \times Far \times Near}{Far - Near}$$

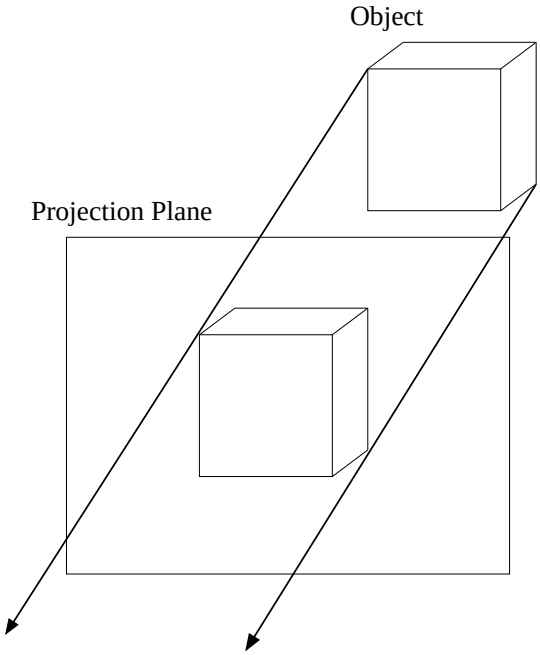

Figure 2.4: Orthogonal Projection

Using these definitions, z^* is considered a pseudo-depth because it is not the true depth to a vertex, but does preserve the depth ordering of vertices in the view volume.

The projection matrix is defined as [53];

$$\begin{bmatrix} \text{N} & 0 & 0 & 0 \\ 0 & \text{N} & 0 & 0 \\ 0 & 0 & \text{a} & \text{b} \\ 0 & 0 & \text{-1} & 0 \end{bmatrix}.$$

Multiplying the point (wP_x, wP_y, wP_z, w) by this matrix yields $(wNP_x, wNP_y, w(aP_z{+}b), -wP_z)$. (Using homogeneous coordinates a scalar multiple of a point is the same point – i.e. if w is not 1). Dividing through by $-wP_z$ (called perspective division) and discarding the fourth component yields

$$\left(N\frac{P_x}{-P_z}, N\frac{P_y}{-P_z}, \frac{aP_z+b}{-P_z}\right) \quad ,$$

which are x^*, y^*, and z^* defined previously. The transformation to the default view volume and perspective projection can be combined into a single matrix.

Techniques to avoid displaying portions of polygons obscured by other polygons are referred to as hidden surface removal algorithms. The painter's algorithm assumes polygons are ordered from back to front when sent to the rendering pipeline similar to the way a painter paints foreground

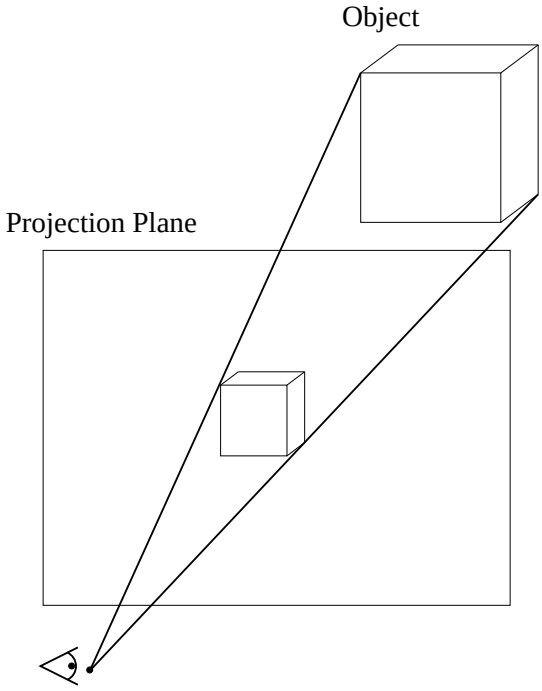

Figure 2.5: Perspective Projection

objects over background objects [2]. The z-buffer algorithm uses pseudo-depth values to determine visibility during rasterization. Rasterization converts vertices of a polygon to pixels using scan line algorithms. As shown in Figure 2.6, the polygon edges are used as endpoints to scan across the interior (the convex polygon requirement simplifies this process).

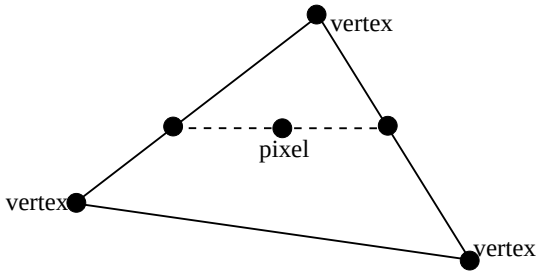

Figure 2.6: Rasterization

The z-buffer algorithm associates a pseudo-depth value in a depth buffer with each pixel in the frame buffer proportional to the true depth. Initially the pseudo-depths are all set to a maximum value, but are updated as pixels are determined. The frame buffer is written to only if the new pseudo-depth value is lower than the current pseudo-depth [53].

Scaling to the default view volume may distort the scene. However, the viewport transformation,

which transforms points from the view plane to the viewport (i.e. the area displayed on the screen), can restore the aspect ratio.

2.3.3 Shading

In OpenGL, and other rendering methods, the intensity, I, of light at a point is governed by the Phong model [71] and is applied for each light source.

$$I = \frac{1}{a+bd+cd^2}(k_d L_d \vec{l} \cdot \vec{N} + k_s L_s (\vec{r} \cdot \vec{v})^\alpha) + k_a L_a$$

Variable d is the distance from a light source and $\frac{1}{a+bd+cd^2}$ is a quadratic attenuation term representing a decrease in light intensity as the distance from the light source increases. Specifying constant, linear, and quadratic terms allows for greater control of the attenuation. The $k_d L_d \vec{l} \cdot \vec{N}$ term represents diffuse lighting illumination which occurs when light interacts with a rough surface and is reflected in many directions. L_d is the diffuse intensity from the light source, $\vec{l}$ is the unit vector of the incident light, and $\vec{N}$ is the unit normal of the surface at the point of incidence. Specular highlights are governed by the $k_s L_s (\vec{r} \cdot \vec{v})^\alpha$ term where L_s is the specular intensity of the light source, $\vec{r}$ is a unit vector along the direction of the reflected light, and $\vec{v}$ is a unit vector in the direction from the point of incidence to the viewer. The parameter α governs the spread of the specular highlight. The ambient light in the scene, L_a, accounts for light after many reflections off of objects in the scene and is approximated as a constant. The k_d, k_s, and k_a terms characterize the surface's interaction with each category of light and together are called the material properties [2]. The Phong model can be applied separately to red, green, and blue components of light.

Three types of shading utilize the Phong model: flat shading, Gouraud (i.e. interpolative) shading, and Phong shading. Flat shading assumes the viewer and light source are located an infinite distance away so that the angle of the incident light ($\vec{l}$), viewing vector ($\vec{v}$), and reflected light ($\vec{r}$) remain constant across a polygon. Assuming a constant normal ($\vec{N}$) across the polygon's surface, the intensity of the light at any point on the polygon is constant so the lighting calculation only needs to be performed once for a given polygon and light source.

To counter the abrupt colour changes encountered in flat shading, Gouraud shading computes the light intensities at the vertices of a given polygon and interpolates pixel colours across the polygon [45]. To create smooth colour transitions between adjacent polygons, the vertices' normals are set to the average of the adjacent polygon normals. Phong shading [71] differs from Gouraud shading by interpolating the normals, rather than the colour intensities, across the polygon. At each pixel the Phong model of lighting is applied.

2.4 Chapter Summary

This chapter introduces the background of the two subfields used in this thesis, SCSPs for selecting camera feeds, and OpenGL for rendering graphics. A SCSP generalizes classical constraint satisfaction problems to the language of preferences. Subsequently, a solution to a problem is not a tuple that satisfies all the constraints, but one that maximizes a preference value (or minimizes a cost). Constraint propagation algorithms, using node and arc-consistency, can simplify a problem, such as a WCSP, to a simpler problem that has the same solution. OpenGL renders polygons specified as points, to the display as pixels. Vertices of polygons are processed in homogeneous coordinates, using matrix multiplication. A Phong lighting model colours pixels shown on the display.

Section 6.2 uses points from the OpenGL pipline to determine the location of objects on the display, to set dynamic constraints.

Chapter 3

Related Work in Virtual Cinematography

Computers can produce realistic looking images, but placing and aiming the virtual camera to acquire aesthetically pleasing images or animations is typically done manually. Specifying low level camera parameters is tedious, especially when the programmer provides them for each time step in an animation. Even with a still image, a frustrating process of trial and error can ensue as the user sequentially guesses where to move the camera.

Some three-dimensional computer games solve the camera placement problem by fixing the camera to the main character's view. This solution, however, fails to take advantage of certain film techniques that have developed during the first half of the twentieth century to which film audiences became acclimatized, even if they do not actively recognize their use [3, 64]. For example, a wide shot showing a far away view of a scene, called an establishing shot, usually introduces a scene. By utilizing these understood conventions, games and other three-dimensional applications can produce more expressive and pleasing output.

Researchers are attempting to solve these problems by creating methods to automatically place and aim the virtual camera. Earlier work concentrates on camera placement, while more recent work desires to instill aesthetics and emotions into the virtual filming process.

3.1 Through-the-Lens Techniques

An early and widely cited paper by Blinn [13] introduces the problem of camera placement and a solution in a restricted environment. Blinn's system automatically positions and aims the camera to generate an animation of a spacecraft during the time it passes a planet. Figure 3.1 shows a frame from the animation. Blinn's system directly computes a transformation matrix to position and aim the camera given the points on the spacecraft and planet, and the distance between the craft and the camera.

To produce a more interesting animation, Blinn describes how to generate transitional sequences that align with the spaceship-planet sequences. These transitional sequences need not have a planet in the background for the entire duration, and they position the camera in such a way that the spaceship and planet align with other sequences already described. Thus the camera can follow a

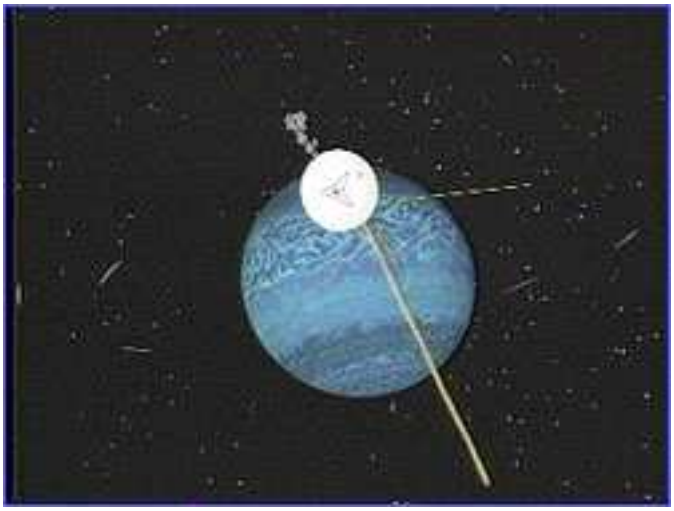

Figure 3.1: Voyager Fly-By (http://www.pbs.org/wgbh/nova/specialfx2/images /voyager.jpeg)

spaceship as it passes a planet, follow the ship as it flies into space and continue as it passes another object such as a planet, moon, or asteroid.

Gleicher and Witkin describe a more general "Through-the-Lens Camera Control" system where the user pins points in the world coordinate system to points in the screen coordinate system [44]. The user selects points on the three-dimensional model to keep at roughly the same position on the screen as the scene evolves in time. The computer solves for the camera position and angle that keeps the pinned points in the same location in the screen coordinate system as the scene changes. Figure 3.2 presents an example of pinning two corners of a cube to the screen.

Figure 3.2: Through-the-Lens example. As the camera moves in the direction of the arrows, the corners of the cube are pinned to their initial screen coordinate positions. The camera rotates and adjusts the level of zoom to achieve this. [44]

Additionally, the user may drag a pinned point to a desired location, and the camera's parameters will adjust over time to move the point to the desired position in the screen coordinate system. An arbitrary number of points can be pinned to the image subject to the number of degrees of freedom. An example exceeding the number of degrees of freedom includes pinning two points of an object to two points in the screen coordinate system and requesting that a third point travel too far off screen to be simultaneously viewed. Conflicts, such as exceeding the number of degrees

of freedom or requesting that a point move in two directions, can either be forbidden by the user interface or handled by distributing the error uniformly using a least-squares solution.

In this more general environment it becomes necessary to solve non-linear algebraic systems for which there is no general recipe. Blinn solves for the transformation matrices directly, whereas Gleicher and Witkin determine the change in world space velocity. Possible methods of acquiring this velocity include interpolating between key frames, or it may be given by the user via mouse movements. The change in world space velocity is mapped to a change in camera velocity and integrated using a discrete time step to determine the camera's position and orientation. Using position feedback, a corrective term reduces errors that accumulate over time due to the integration method.

Christie and Hosobe extend Gleicher and Witkin's idea to cover camera, lighting, and objects under a common technique they call Through-the-Lens Cinematography [19]. As in the work from Gleicher and Witkin [44], the user pins points from the world coordinate system to points in the screen coordinate system to indirectly control the camera. Additionally, the user can specify that points translate or rotate objects in the world coordinate system. Christie and Hosobe provide an example of pinning an airplane's wing tip while moving the other wing tip, thereby rotating the plane about the fixed wing tip.

Points in the world coordinate system are virtually attached to their destination by fictitious springs. A deviation from a spring's rest length produces a force under Hooke's law ($F = -k\Delta x$), which is mapped to acceleration using Newton's second law of motion ($F = ma$). The user can adjust the value of m to achieve the desired rate of acceleration at a point.

In addition to points, the camera and objects are controlled via lines, circles, or splines. For example, the user could keep an airplane wing in the output image by drawing a line along the top of it. These more complicated primitives, however, are mapped to one or more individual points before the virtual springs are attached.

Manipulating lighting is similar to manipulating objects or the camera, the user moves the center of a specular highlight to the desired destination and the system moves the lights to produce the highlight in the specified location.

3.2 Constrained Solutions

Predating the through the lens techniques done by Christie and Witken, Drucker and Zelter introduce a framework called CamDroid for specifying camera constraints [34, 35, 32]. Rather than providing constraints via a virtual lens, constraints are encoded in camera modules. Modules are software encodings that represent shots in cinematography and are programmed using a text editor or a visual editor, where constraints can be dragged and dropped into a module. When filming with

a virtual camera, sequential shots may align and appear as a continuous shot belying the sequential camera modules controlling the output. Multiple constraints may make up an individual module. For example, one constraint may insist that the camera remain at a specified height, while another constraint directs the gaze of the camera. A constrained optimizing solver combines constraints to produce the final camera parameters for a particular module [32]. Branching connections between modules governs the order of modules selected to produce the final animation.

An example application of CamDroid is a virtual painting museum. A screen shot is shown in Figure 3.3. The system plans a path through the museum to visit all user requested paintings. The path satisfying the constraints is constructed as follows. The order of room to visit is determined by an A* search where edges represent the rooms' doors and nodes represent rooms [34]. The rooms containing paintings to visit are ordered, subject to the adjacency conditions, using an exhaustive search. The path between doors within each room is constructed by following the distance gradient within each room. The gradient is constructed as follows. Each room is projected onto the two-dimensional floor and discretized into cells where each cell is open or contains an obstruction. For each open cell, a distance from the originating door is computed travelling along any of 32 directions of movement (recall that a Manhattan distance uses four directions: N, E, S and W. A 32-way connectivity map can use the outer 32 cells of a 9×9 grid). A discrete path from the exit to the entrance can be constructed by descending the distance gradient.

Figure 3.3: Camera Shot of the Virtual Museum [34]

To avoid grazing paintings a simulated circular repulsive gradient emanates from each painting, and is combined with the distance gradient. To avoid becoming trapped in local minima, a breadth first search uses the gradient information to construct a path from entrance to exit. To create smoother movement, a spline is fitted to the cells that make up the path using a least squares curve fitting method. Additional tangency constraints on the spline ensure the path goes through doorways in a perpendicular fashion.

Less complicated camera constraints ensure the camera remains at a constant height, a sensible up vector is maintained, and the camera is aimed at paintings, when available, and along the direction of motion otherwise. Each constraint can be encoded into a module to be included in the

overall system.

In a subsequent paper, CamDroid is applied to filming a conversation and a football game, as additional examples [35]. When filming a conversation, two (imaginary) vertical lines divide the screen into thirds. One constraint specifies that the person facing the camera should be along the two thirds line while another constraint specifies that the person facing away should be along the one third line. Other constraints specify that the world should be aligned up, the camera should be close to aimed directly onto the character facing the camera, and the field of view should be between twenty and sixty degrees. The system can identify which character is speaking and films, the conversation by interconnecting two camera modules - one for when each character is speaking. As for filming a virtual football game the camera can track individual characters or the ball, or orbit about a player.

A paper from Bares *et al.* also presents a constraint based camera positioning system [5] using a subset of the constraints available in Drucker and Zelter's work [34, 35]. Users may specify vantage angle, viewing distance, occlusion avoidance, and require that the camera remain confined to a room. Specifications have optimal settings and acceptable ranges. The constraint system first eliminates ineligible regions of the search space. For example, specifying a vantage angle eliminates other unsatisfactory angles. The union of the resulting space with eligible space, specified by the viewing distance, further constrains the solution space. From the refined solution space an individual specification, or an average of the given specifications, is selected [5]. For example, if there are three optimal vantage angles specified for three individual objects, one of these vantage angles could be selected subject to the condition that it falls into an established region of acceptable solutions. In a subsequent paper from Bares *et al.* [7] the space is heuristically and recursively searched. A finite number of candidate shots are proposed and evaluated with the best shots recursively refined.

If it is impossible to simultaneously satisfy all requirements, then the weaker requirements are removed for a solution satisfying the most important constraints. If no solution can be found even after removing weak constraints, then the output visualization may employ multiple views. As an example, suppose two characters are too far apart to simultaneously satisfy the viewing distance requirement. The system may generate a far camera shot of the two characters and then provide aninset of each character in the upper corners of the image.

To facilitate entering constraints, Bares *et al.* use a storyboard system that acts as a graphical user interface [7, 6]. Users provide optimal, maximum, and minimum specifications for various attributes and indicate weaker constraints. (As mentioned above weaker constraints can be ignored if needed to arrive at a solution.) For example, the size of a character in the output image is set by manipulating a large and small box to indicate the allowable dimensions. The range of viewing angles for an object, field of view, and occlusion are similarly defined. Partial occlusion allows

a degree of overlap between objects and provides a sense of depth in the image. Constraints are exported to the constraint solver to determine the camera parameters.

3.3 Camera Positioning using Weighted Constraints

A difficulty with the constraint approach in Section 3.2 occurs when the problem is over-constrained so that no solution is possible. Bares' work allows removal of weaker constraints to find a solution, but it may be better to satisfy multiple weaker constraints than one strong constraint at the expense of multiple weaker ones. With this in mind, Bourne *et al.* introduce the use of weighted constraints to position a virtual camera [16]. They determine the camera's position (x, y, and z) through an increasingly focused search to maximize the combined weighted preference.

The soft constraints, or preferences, to satisfy are *height* (length), *distance* (length), and *orientation* (degrees) which are assigned an increased cost as they deviate from their target positions. Weights assign importance to each preference and are normalized to account for differences in scale. The overall preference is the weighted average of the included preferences. A *coherence* preference discounts solutions nearby the current camera position to avoid the camera jumping to more distant solutions. When desired, an *occlusion* preference can prefer a line of sight to the main subject by determining intersections between four rays from the target to the camera with objects in the scene (using similar calculations to ray tracing [85]). Hard constraints, as presented in Section 2.1.1 and used in Section 3.2, can be approximated by using a very high weight.

A *sliding octree solver* (the spatial data structure is shown in Figure 3.4) determines where to position the camera. Given the distance the camera is permitted to move each frame, eight positions in the space are considered. Upon determining the best candidate position using the preferences given, the permitted camera movement is reduced and the octree solver determines a finer position starting from the current solution. Thus the sliding octree solver can iteratively determine the most preferred camera position, even when the constraints can not all be fully satisfied. Where hard constraint approaches may yield multiple solutions, the soft constraint approach can order the solutions by preference.

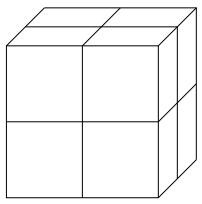

Figure 3.4: The octree spatial data structure. (Figure 4 from Bourne *et al.* [16])

Different weights on the preferences yield different filming styles, similar to idioms that will be

discussed in Section 3.4. Defining the weights for a particular filming style defines a camera profile, and combining camera profiles enables filming cinematographic sequences. Figure 3.5 shows example state transitions between camera profiles for filming a conversation. When multiple characters are filmed the target is a weighted average between the characters, with the speaker given a higher weight.

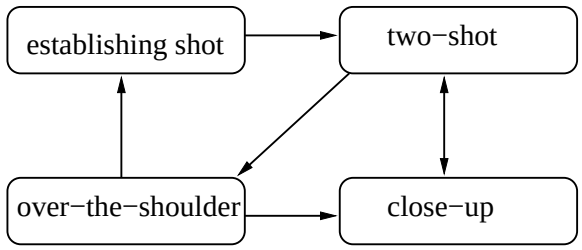

Figure 3.5: Example state transitions between camera profiles. (Figure 11 from Bourne *et al.* [16])

A high enough weight on the coherence preference prevents jumping abruptly to a new position when changing camera profiles.

Alam and Goodwin build on the weighted constraints of Bourne *et al.* by determining isocurves that bound regions that satisfy a constraint [1]. They present a 2D example using a total weight cost for a problem given as follows:

$$k_1\rho_1 + l_1\theta_1 + k_2\rho_2 + l_2\theta_2.$$

Here ρ_1 is the difference between the desired position and current position with respect to visual constraints, and θ_1 is the difference between the desired angle and current angle with respect to visual constraints. Variables ρ_2 and θ_2 are the difference in position and angle of frame coherence constraints, to prevent the virtual camera from moving too quickly in position or angle. Frame coherence is also discussed in Section 3.7. Variables k_1, l_1, k_2, and l_2 are weights that determine how important each constraint is to satisfy.

If the visual constraints have higher weights than the frame coherence constraints then an isocurve bounds the region of acceptable visual constraint solutions, given by $\rho_1/\rho_{10} + \theta_1/\theta_{10} =$ constant, where ρ_{10} and θ_{10} are the ranges of acceptable values for ρ_1 and θ_1. If the constant is 1 then the isocurve bounds the region that contains positions that satisfy the visual constraints. If the constraint is greater than one then there is no solution that satisfies all the constraints, and a solution to return lies along the isocurve. If the frame coherence constraints have higher weight than the visual constraints, then the isocurve is made from ρ_2 and θ_2. A binary search finds a solution with minimum total weighted cost in real-time.

3.4 Encoding Filming Styles

Drucker *et al.* [34] mention the problem of cuts between shots, which is partially addressed in the filming a conversation example [35]. This problem is dealt with more generally in He, Cohen, and Saliesins' work [49] where they present a system of hierarchical *idioms*. Each idiom prescribes the sequence of shots for filming a scene. These idioms are alluded to in cinematography books as general filming instructions for aspiring cinematographers [3, 64]. For example, a conversation between two actors may start with an establishing shot and then alternate between the two actors, with a far shot used periodically to re-establish the scene. Variations on the rules of an idiom lead to a new idiom, which represents a different style of filming.

In software implementations, each idiom is encoded as a finite state machine (FSM) where transitions represent a new shot in filming (i.e. a cut). Conditions on transitions govern the cuts between shots, such as the maximum or minimum duration of one shot or a change in events. An exception mechanism can wrest control away from the regular flow of transitions through an idiom, which is useful for dealing with occlusion. When an actor of interest becomes occluded, a timer is started. When the timer exceeds a threshold, the camera cuts to another shot with an unoccluded view of the actor. A FSM can be thought to represent a style of filming, in the sense that it encodes how a human director changes from one camera to another.

Hierarchical organizing idioms help select which idiom to use for a particular shot. Idioms lower in the hierarchy are more specific than idioms higher in the hierarchy, so the lowest applicable idiom should be used for filming. As an example, consider an idiom for filming two people. Knowing nothing else may dictate the use of a general two actor idiom. If it is known that the two actors are talking, however, then a more specific talking idiom can be used. Similarly, if the two actors are fighting then a fighting idiom can be employed. Using a hierarchy ensures that there is always an idiom available to govern filming.

Associated with each state is a camera module specifying the virtual camera's parameters. The authors' implementation uses 16 camera modules (apex shot, internal shot, external shot, etc.), which are fairly rigid in their implementation [49]. However, they express interest in a more flexible constraint solving technique, similar to Drucker *et al.*'s implementation, as future work. A constraint implementation could differ from most other automated camera placement work since the camera modules can exert subtle changes on the objects in the scene. For instance, when filming two actors close up the actors are moved closer to each other resulting in a better looking scene. This is similar to Mascelli's cinematography advice to remove distracting objects from a scene in such a way that the audience doesn't notice [64].

Christianson *et al.* [18] also use idioms, but their paper focuses more on users constructing idioms using a Declarative Camera Control Language (DCCL). Structures in DCCL are constructed from

four basic primitives: fragments, views, placements, and movement endpoints. A fragment is a part of a shot where the camera's velocity is continuous. This means the camera is stationary or moving in a simple motion, such as an arc. When a shot is made up of multiple fragments the end and start of adjoining fragments must align. With some fragments, movement endpoints indicate how much of the screen the actor covers while they are moving. Views can be extreme, close-up, medium, full, or long and the placement of the camera can be internal, external, parallel, or apex as shown in Figure 3.6.

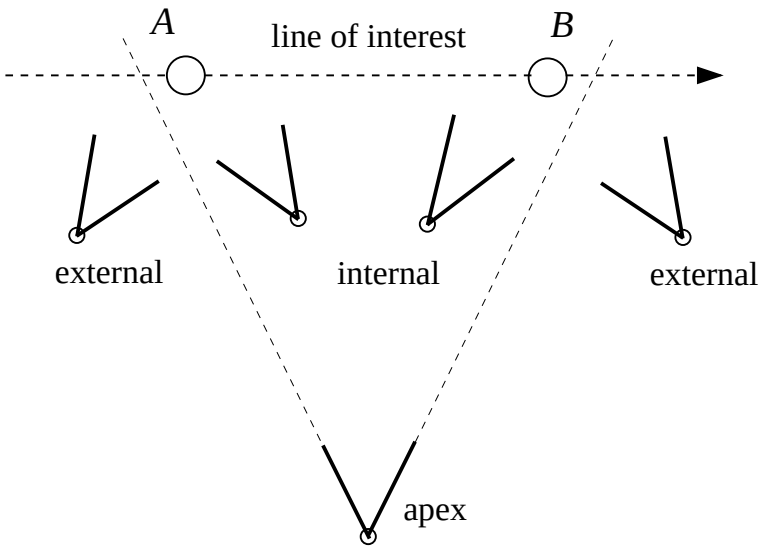

Figure 3.6: Camera placement is specified relative to "the line of interest". From Figure 1 of Christianson [18] which is an adaptation of Figure 4.11 in Arijon [3].

The system to compile and run DCCL is called "The Camera Planning System (CPS)" [18]. Given an animation trace (a sequence of the positions and orientations of virtual characters) CPS determines the activity being performed. In the example application, twelve activities defined include moving, turning, looking, and stopping for which one or more idioms are assigned as possibilities for the virtual camera's parameters. Thus, given an animation trace, a tree structure can be constructed which contains all possible relevant idiom combinations. A "Heuristic Evaluator" selects the branch of the tree for final rendering based on four criteria: smooth transitions between fragments, avoiding crossing the imaginary line between actors, avoiding very long or very short fragments, and avoiding fragments in which the camera pans backwards. While the exhaustive approach taken in the work done by Christianson *et al.* is intractable in general, the number of idioms per action is kept low resulting in a low branching factor (only four of the twelve activities have two possible idioms, the remainder have only one) [18].

3.5 Other Camera Selection Mechanisms

Assa *et al.* present a method to select a camera from a set of cameras based on human motion in a scene [4]. A sequence of selected camera shots generated by their system is shown in Figure 3.7. Their approach uses the correlation of 3D input data with the pixels seen through a particular camera feed and selects the camera feed with the highest correlation. Their method does not require specifications from a human, but rather selects the viewpoint that displays the most motion.

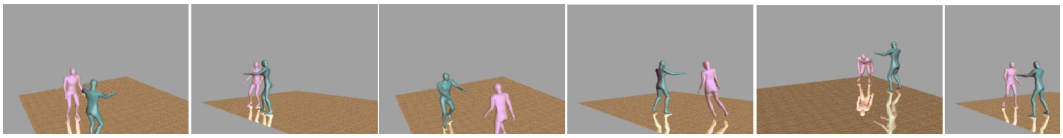

Figure 3.7: Camera viewpoint selected to show motion [4]

Passos *et al.* train a neural network to select a camera feed in a car racing game as shown in Figure 3.8 [4]. While a human player controls the race car, another human director selects which view to display from among three camera feeds. This input is used to train a neural network, which Passos *et al.* report is able to replicate the human director with 100% accuracy.

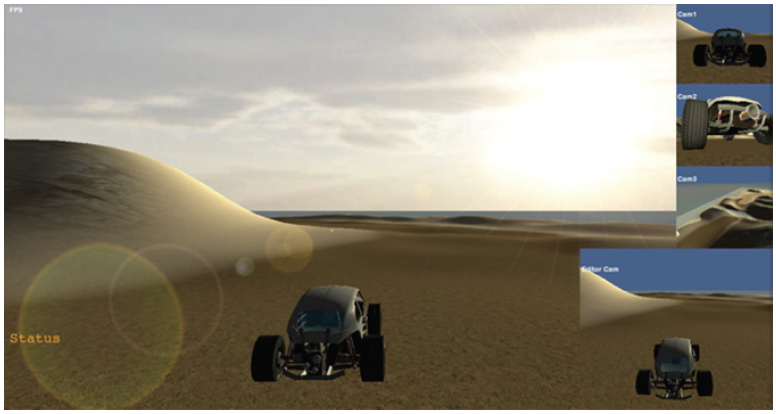

Figure 3.8: Car racing game to train neural network [69]

Lino *et al.* present a cinematography system which includes a camera cutting aspect [62]. Their work uses director volumes to determine where to position the camera and map candiate camera positions, such as a location having full or partial visibility of key subjects. Their camera planning approach utilizes frame coherence, and their system can handle occlusion of non-moving objects. The camera selection mechanism is based on idioms, and includes narrative elements modelled using semantic tags. The semantic tag includes things such as the following: a key subject stands up, a key subject talks to another subject, show dominance of one subject over another, show conflict,

and show isolation. The system executes in real time and uses the semantic tags when selecting an idiom.

3.6 Automatically Determining Camera Target

In the work described above, the user directly or indirectly specifies the target object to film and assumes that the angle to film the target object is reasonable. CamDroid films the paintings in a virtual museum from a path through the museum, and Bourne's camera system attempts to film the player's avatar from an unoccluded position [34, 16]. Some recent work attempts to automatically determine the object to film and the best angle at which to film it.

Turkay *et al.*'s work uses an information theoretic approach to determine subjects to film in a crowded scene [82]. Their system films characters walking around and films interesting action, where 'interesting' means 'unexpected'. A character's future location is predicted based on their past action using their position, direction, and speed which are projected onto a two dimensional plane (i.e. the ground). A *historical probability function* predicts future actions using a weighted linear combination of past actions from a discrete number of time steps, which is a user tunable parameter. Players' expected movements are grouped into cells on a grid and compared to actual movement in each cell. Cells with a difference between the predicted movements and actual movements exceeding a threshold are considered for filming. When no cell exceeds the threshold, the camera makes a tour over cells where players move together, keeping track of visited cells to avoid excessive repetition.

Work by Vazquez determines a good viewpoint to film an object using *view stability* on depth images [83]. A view is stable if views from nearby viewpoints are similar. Two viewpoints are considered similar if their depth images are small when compressed together. To consider 320 positions about an object currently, using 256×256 gray scale images, requires approximately 10 minutes of calculation [83], and is largely independent of the complexity of the model. Future work with adaptive or a hierarchical approach may reduce the computational time. Vazquez's paper argues that the views generated by the system are close to preferred views. Examples are shown in Figure 3.9.

3.7 Balancing Camera Placement and Frame-Coherence

With the camera positioning systems previously described, the virtual camera may move erratically as it seeks out the optimal position, particularly with a constrained solution ([5] for example). In a paper by Halper, Helbing, and Strothotte [47] a balance is sought between optimal camera placement and smooth camera motions, referred to as "frame-coherence". The system respects frame-coherence by following four principles: solving from an existing state, parameter relaxation, occlusion avoidance, and look-ahead algorithms.

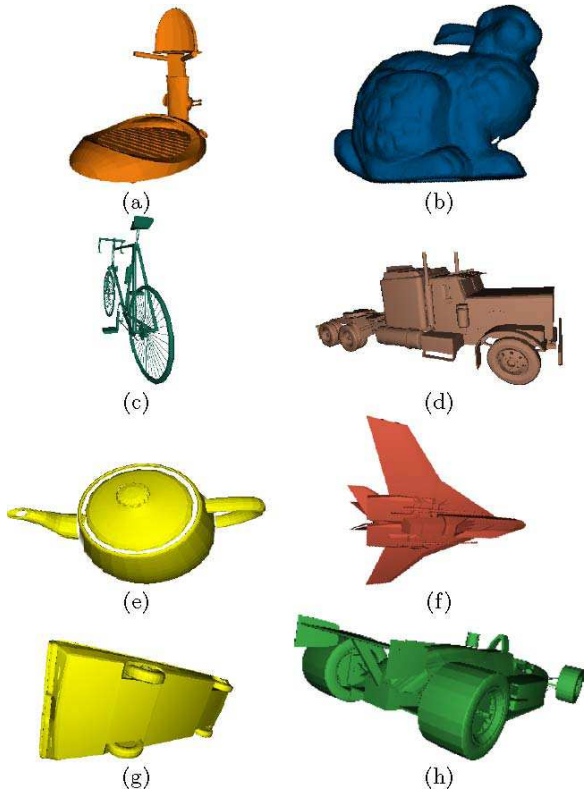

Figure 3.9: Viewpoints selected using view stability [83]

The constraint solver determines camera position by starting from existing camera parameters to find nearby solutions. From its current state the camera's parameters are adjusted closer to a goal state in parameter space, such as position or angle, solving constraints in an order to minimally influence the output of previous solutions. For example, adjusting the size of the subject by moving the camera closer or further away does not alter the angle used to aim the camera.

Similar to Bares *et al.*'s work [5], each goal has a tolerance region which contains acceptable, but suboptimal, solutions. Parameter relaxation [47] refers to this tolerance region where the optimal settings of the constraints are not strictly obeyed. If a parameter of a proposed solution falls outside of the tolerance region it must be placed at the edge of the region. When too many parameters fall outside their tolerances a transitional cut may be selected.

To avoid occlusion, Halper *et al.* use a hardware accelerated variation of a technique described by Feiner and Seligmann [39]. The technique [39] identifies regions from which all desired targets (subjects and objects) are visible. An example in two dimensions is shown in Figure 3.10. Similar to algorithms for projecting a scene onto the image plane in Section 2.3 (as in z-buffer algorithms [2, 53]), areas of occlusion are determined as shown for one target on the left side of Figure 3.10. As additional targets are added, the number of occluded areas increases (right side of Figure 3.10).

Remaining non-shadow areas contain unoccluded views of all subjects, although it may not be possible to simultaneously aim the camera at all subjects (for example, if the camera is placed between the two subjects).

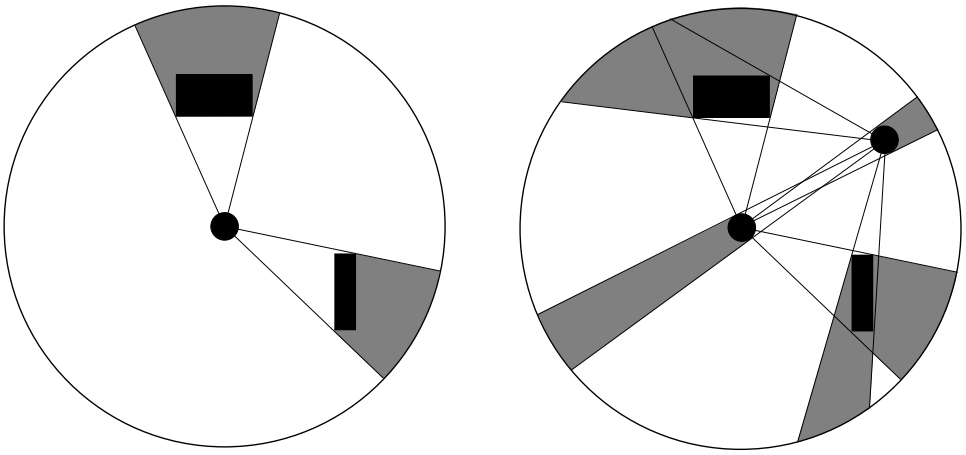

Figure 3.10: Potential Visibility Regions. White areas have an unoccluded view of the subject(s).

Feiner *et al.*'s technique of determining occluded areas differs by partitioning space into a binary spatial partition (BSP) tree containing shadows or unoccluded views [39]. Problems with this technique arise when using graphics hardware rendering algorithms to accelerate the approach, since the shadows must be cast onto something. Locations on the periphery can be selected, but this leaves some valid locations without a chance of being selected. Additionally, the user is unable to rank potential solutions and, in some cases, the accelerated technique misses finding a solution when one exists. Halper *et al.* present the example, shown in Figure 3.11, where the camera on the periphery is unable to find an unoccluded view [47]. An unoccluded view exists at point **P** in Figure 3.11, but this point is not on the periphery.

Halper *et al.*'s solution is *Potential Visibility Regions* (PVR). Shadows are cast onto user constructed polygons roughly facing the object or objects of interest. Brighter PVRs indicate a preference for camera placement. After the camera is positioned at each target and shadows are cast by the occluding objects onto the PVR polygons, the brightest remaining area on a PVR indicates the preferred unoccluded view of all targets.

PVRs may be generated at a lower resolution than the final output image, and the occluding geometry may be simplified in order to increase the algorithm's speed. This results in the potential for the algorithm to be applied in real-time using a graphics chip. The cost of solving for multiple subjects is linear with respect to the number of subjects.

Look-ahead algorithms, which assume the future locations of objects based on their trajectories and accelerations, also improve frame coherence. The constraint solver is applied to this predicted

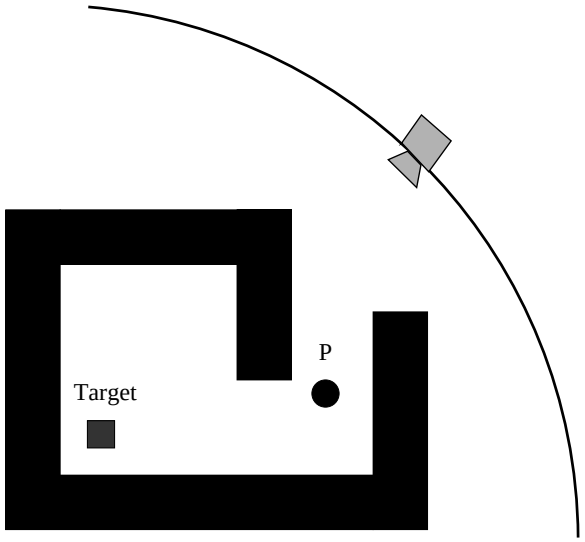

Figure 3.11: Camera is unable to find an unoccluded view (for example, at position P) of the target.

state. Deviations from the planned camera trajectory may occur, but overall the prediction method increases smooth camera movements (at the cost of added computation). Alternatively, if the cost of object prediction is too great, the constraint solver may bias its solutions to ones that fall near the predicted path of just the camera rather than considering future locations of objects.

3.8 Encoding Emotional Content

Human cinematographers use camera angles and other cinematographic techniques to enhance the emotional content of a film shot. Including emotional content via filming techniques is, for the most part, ignored with virtual cinematography systems. Some researchers, however, have made efforts to infuse emotional content into automated virtual compositions.

Work from Tomlinson, Blumberg, and Nain describes a filming system incorporating emotions and motivations to enhance the filming style of their virtual cinematography system [81]. Their reactive and interactive system uses happy, sad, angry, surprised, fearful, and disgusted emotions to modify the filming parameters. For example, a happy state may include bouncy and swooping camera motions with a brightly illuminated scene and many close-up shots. Conversely, a sad state may include slow sweeping arcs and dim lighting. Occlusions are avoided by moving the camera forward until the camera passes the occlusion.

Their approach is to consider the camera as a special actor in a virtual environment, along with other virtual actors. They call the camera "CameraCreature". Each actor has a simple emotional model that modifies the CameraCreature's emotions, weighted by the actor's importance,

the intensity of the emotion, and the amount of recent screen time of the actor.

Motivations select which type of shot is called for. For example, a motivation termed *DesireToEstablish* causes a wide establishing shot. This particular motivation rises at a constant rate, but is self-inhibiting and causes periodic establishing shots to occur, unless a more pressing motivation takes precedence. *DesireForTwoShot* is the default motivation having a constant, reasonably high value. It frames two characters in the shot as shown in Figure 3.12.

Figure 3.12: A two shot framing. [81]

Characters may request shots which are incorporated as a motivation in the CameraCreature. Two such character motivations are *DesireForCloseUp* and *DesireForActionShot* which are requested when an actor determines their current activity deserves a close-up or action shot respectively. To prevent frequent cuts between shots, or between two shots with close motivational values (which the authors term "behavioural aliasing"), a motivation must have twice the intensity as the current motivation to take over. Their system is demonstrated using a simple game in which the main character is assigned a high importance so the camera only cuts to another character when they are doing something very important.

Kennedy and Mercer present another method of incorporating emotion into a planned virtual filming sequence [54]. Their system modifies the emotional content of an animation by merging a knowledge base with a user's directions. The user defines characters, objects, and actions while the computer adjusts the camera and lighting (the user can over-ride the computer's choices). A planning agent (written in LISP) uses the knowledge base to generate plans to control the camera and lighting. For example, a scary scene may use dim lighting and close-up camera shots. In terms of camera cuts, the user may specify fastpaced cuts or slowpaced cuts resulting in many short cuts, or fewer long cuts, respectively. Images are generated using the POVray ray-tracer. Figure 3.13 shows a happy rendering, while Figure 3.14 shows a scary rendering of the same scene. In the

happy rendering the computer has chosen a more distant camera position from the subject and front lighting, while the scary rendering uses a closer shot and back lighting.

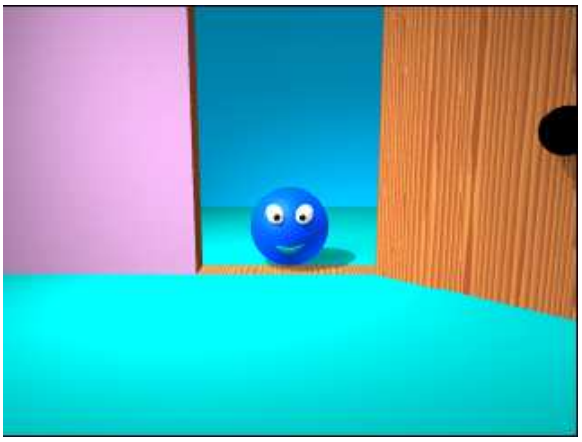

Figure 3.13: Two Variations of the Same Scene with Different Emotional Content: Happy Entrance [54]

Figure 3.14: Two Variations of the Same Scene with Different Emotional Content: Scary Entrance [54]

3.9 Occlusion

Occlusion is typically ignored or avoided (i.e. by moving the camera to an unobstructed viewpoint). Through-the-lens techniques ignore occlusion, and instead depend on the user to avoid specifying camera movements resulting in occluded views. Constrained solutions search for unoccluded locations to place the camera using either explicit or implicit occlusion constraints. For their virtual cinematographer, He *et al.* use a timer to reposition the camera if a subject becomes occluded for

too long [49]. Bare's approach allows the user to specify tolerances for occlusion and permits a requirement for one object to occlude another by a certain amount [5]. When avoiding occlusion, however, Bare's solution repositions the camera.

Seligmann and Feiner also deal with occlusion when displaying intent-based three-dimensional illustrations [39, 78]. Although not intended for virtual cinematography, their approach suggests an alternative to merely avoiding occlusions when generating three-dimensional illustrations to help follow textual instructions. As such, the user generating the illustration can specify an unoccludable object, an action to perform, and style rules to govern the generation of the illustration, such as how much of the item must be displayed in order to maintain context. Actions to illustrate include pushing, pulling, loosening, lifting, and inserting. Given this input, the intent-based illustration system (IBIS) generates an appropriate illustration [78], possibly with meta-objects, such as an arrow to show which way to turn a knob.

A potential problem, when generating an illustration, is that the unoccludable object may be occluded from all viewing positions, such as replacing a battery inside an electronic device. To solve this problem, Seligmann and Feiner introduce cutaway views and ghosting [39]. Ghosting displays the occluding object as semi-transparent so the user can see through to the unoccludable object (such as a battery). Cutaways remove the part of the object that occludes the unoccludable object. Figures 3.15 and 3.16 show a ghosting and cutaway view of a battery inside a radio. Multiple passes of the z-buffer algorithm create cutaway views. First, a mask that covers the unoccludable object is rendered and assigned the minimum distance to the screen. When using z-buffer algorithms ([2, 53]) this ensures that no occludable object will be drawn to the region where the unoccludable object will be rendered, since occludable objects will be deemed to be behind the mask. Next the occludable objects are rendered. Lastly the distance is reset to the maximum distance possible and the unoccludable object is rendered. Since the distances have been reset, the unoccludable object will be fully visible.

There are potential problems with this approach. One is that the context of the unoccludable object may be lost since occludable objects behind the mask will not be rendered, even if they are behind the unoccludable object. Another problem is dealing with self occlusion, or multiple unoccludable objects. To maintain context the masks are typically only slightly larger than the unoccludable object. To deal with multiple unoccludable objects and self occlusion, the IBIS system can generate a composite illustration where inserts show the illustration from multiple viewpoints.

3.10 Cinematography in Computer Video Games

With the advent of three-dimensional computer games, the choice of camera placement can strongly affect the enjoyment of the game. As stated by Christie *et al.* "Camera systems are increasingly

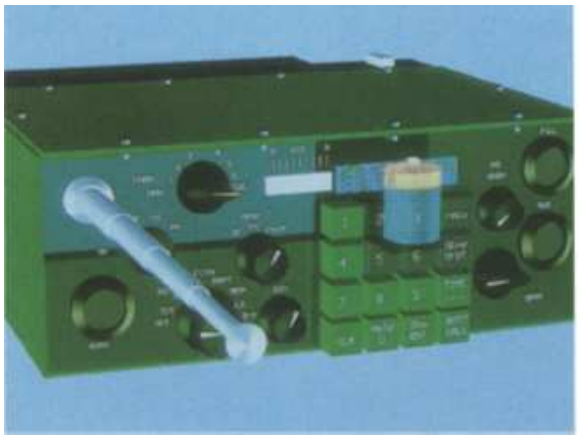

Figure 3.15: Illustration Showing a Battery inside a Radio: Ghosting [39]

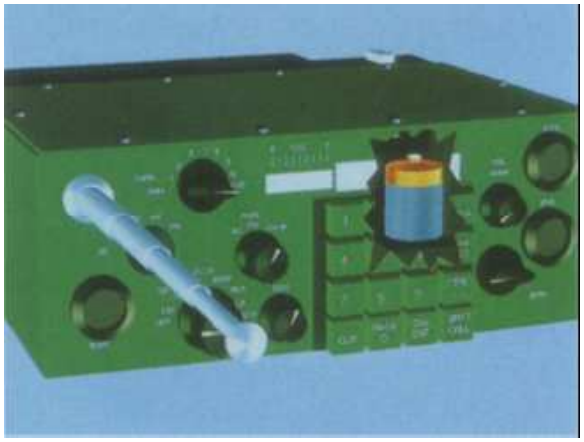

Figure 3.16: Illustration Showing a Battery inside a Radio: Cutaway View [39]

becoming decisive elements to the success of computer games." [20]. They discuss three main types of camera placement in video games: first person, third person, and action replays [20, 19]. A first person view makes the camera placement problem trivial (since the camera view is the player's view), with the most common genre to use this type of view being the *first person shooter*. A downside of a first person view is a rigid fixed camera position. Additionally, first person viewpoints are associated with a lower player-character identification than a third person viewpoint [47]. A third person viewpoint increases the likelihood of the player taking on the role of his or her character. With rigid camera positions, to ensure a correct shot showing all the necessary character interactions, the game must be limited to pre-defined scenarios [47] [1].

A third person point of view typically tracks the main character from a fixed set of positions

[1] Halper *et al.* don't provide an example, but LucasArts' Monkey Island (1990) would be an example where the avatar interacts with other characters that are placed so as to present a good framing for the camera.

relative to the player [20, 19]. Both Christie *et al.* and Halper present the Tomb Raider series [37] as an example of successful games using this viewpoint. However, the options for camera placement become limited in tight spots, resulting in unsatisfactory views. Christie and Olivier give the example shown in Figures 3.17 and 3.18 from Tomb Raider: Angel of Darkness. When the heroine's back is against the wall the camera is forced to move in front of her, resulting in an awkward view for the player. In Figure 3.17 the player must aim facing the avatar, where in Figure 3.18 the player's viewpoint is more closely aligned with the avatar's aim. Halper also mentions the problem of awkward views in close spaces in the Tomb Raider series [47].

Figure 3.17: Camera Positions in Tomb Raider: Angel of Darkness: Bad Camera Position [21]

Full Spectrum Warrior (Pandemic Studios) is a game that gives more consideration to camera positions as described by Christie *et al.* [20]. For example, camera occlusion is avoided through the use of ray casting to prevent views that are blocked by objects. When the camera path is forced through a wall or obstacle, the camera jumps to the scene beyond the wall to avoid passing through solid objects. Other games that have received praise for their "exceedingly competent" [81] camera systems are Zelda and SuperMario64 (both for Nintendo 64). With the exception of fear, however, video games have failed to incorporate emotional content into their filming style [81].

Full Spectrum Warrior, and other games, show a progression in the use of cameras in games, but as noted by Christie and Olivier, currently the use of automated editing and cinematographic techniques in games is rare, and where apparent, it is implemented using ad-hoc techniques [19]. Some games solve the camera placement problem by having the player act as the camera man in addition to playing the game. As an example, in The Sims 2 [38] the user selects actions with the left mouse button, and rotates or translates the camera when pressing the middle or right mouse

Figure 3.18: Camera Positions in Tomb Raider: Angel of Darkness: Good Camera Position [21]

buttons respectively. To avoid camera occlusion the user can specify whether walls in a house are shown, not shown, or use a cutaway view based on the camera angle (there is no solution in the game to objects or characters occluding the camera).

Action replays, used heavily in modern games, highlight significant events such as a car crash in a driving game, or a goal in a football game [20, 19]. In an action replay the player temporarily becomes a spectator, either to emphasize a recent event, or to summarize the events in a game to bring a player up to speed. Drucker and He see an expansion of the spectator aspect of computer games [33]. Just as there are many spectators of a sporting event, there could be online spectators of multiplayer video games, either to preview a game or merely as entertainment. As a spectator, the camera position need not be one of the players' viewpoints. For example, in a war game a spectator could be given a strategic hill-top view, while the players may be restricted to a first person ground troop view.

3.11 Chapter Summary

Automated systems exist for placing, moving, and selecting a virtual camera in a virtual world. Through-the-lens systems enable a user to specify points to pin to the screen, while constrained approaches specify camera constraints to satisfy. Some systems avoid over-constrained problems, while others drop weaker constraints, or maximize the weights of the satisfied constraints. Selecting a camera can be specified in idioms, which are encoded in a hierarchical finite state machine. Other methods of selecting a camera include correlating motion with pixels on the screen, or training a

neural network. Some systems incorporate emotional content into their filming style by adapting the lighting, camera cuts, or camera angle to correspond to an emotional state. Two important aspects of virtual cinematography include frame coherence and occlusion. Frame coherence prevents the camera from rapidly moving to a position far from the current position. Occlusion avoidance maintains an unobstructed view of the actors, based on a timer that counts occlusion duration, or by moving the camera to an unobstructed viewpoint. An area not fully explored in previous work, which will be addressed in the remained of this thesis, is camera selection in dynamic scenes.

Chapter 4

Problem Domains

To test camera selection, two virtual environments are constructed: a spectator perspective hockey game and a third person perspective maze game, shown in Figures 4.1 and 4.3.

4.1 Hockey Game

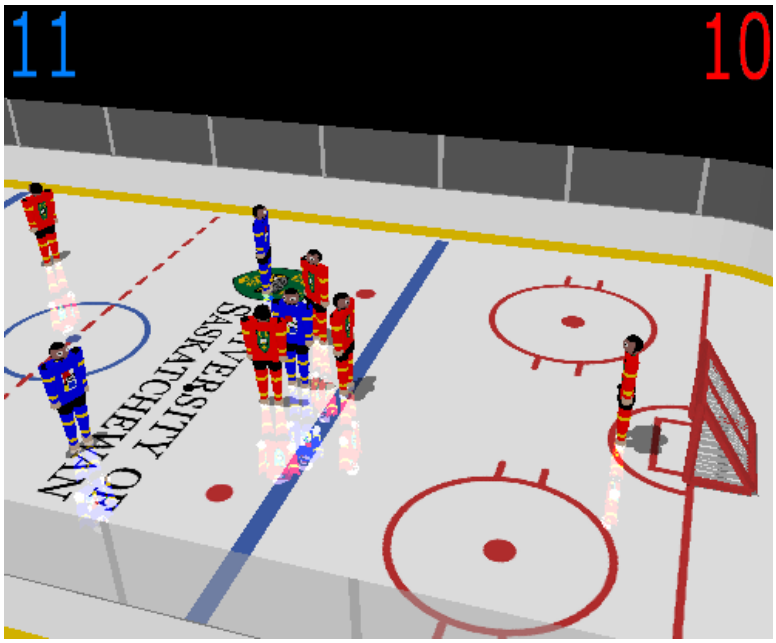

Figure 4.1: Screen Capture of Hockey Game

Ten cameras are placed on one side of the rink, as shown by the white camera men in the two views in Figure 4.2. These cameras were chosen to approximately cover the entire ice rink with some cameras having closer views and others having a view further from the rink [1]. The cameras are placed only on one side of the rink to avoid crossing the line of interest, similar to a broadcast hockey game. The line of interest is an imaginary line dividing the scenes in two; crossing the line

[1] Using ten cameras is arbitrary, but was chosen to correspond to initial manual tests of the simulator where a user selects a camera by pressing the number keys 0 through 9.

of interest is typically considered dissorienting to the viewer [3, 64]. Nearly every part of the rink is visible from at least one camera, but in some parts of the rink, such as near the closer boards, the puck is not visible to any camera. Changes between cameras are abrupt. Such changes are called jump shots [46], and correspond to usual camera transitions in a broadcast hockey game.

Another configuration of the system allows for an arbitrary number of cameras, randomly placed around the rink. This is useful in Chapter 7 when measuring the time needed to select a camera feed based on the number of cameras available.

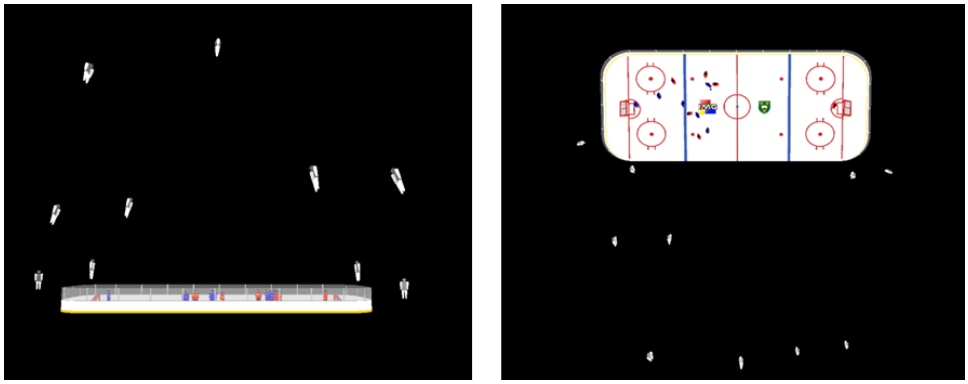

Figure 4.2: Side View and Top View Showing Camera Placement in Hockey Game

The logic controlling each hockey player is deliberately simple. On each team, the player closest to the puck (not including the goalie) moves toward the puck while the other players move toward predefined positions depending on whether the puck is in the home, middle, or visitor region of the ice. Goalies are placed on a hemisphere in the net area facing the puck. When the closest player reaches the puck one of three actions are taken.

- If the player is in the opponent's end of the rink, then the player shoots the puck toward the opponent's net.
- Otherwise, if there is a teammate closer to the opponent's end the player passes the puck forward to the closest teammate that is closer to the opponent's end of the rink.
- Otherwise the player bumps the puck forward to the opponent's end of the ice.

If the goalie stops the puck, then the goalie holds the puck until it is reached by the closest non-goalie player. If the goalie misses a shot on net a goal is scored when the puck passes the goal line. The players skate toward pre-defined positions for a new face-off, and the physical simulation of the puck continues for 200 frames before reseting to the face-off position.

Player motion is defined as follows. Each player can change their acceleration, which affects their velocity and thus their position on the rink, assuming Newtonian mechanics. A player's acceleration

is governed by their distance from their desired position. This causes players to accelerate quickly and slow as they reach their desired destination. The puck also has velocity, which effects position, and friction and gravity influnce the puck's velocity. The puck bounces after intersecting with a board by reversing the puck's velocity corresponding to the board's normal. This means the puck's z direction is reversed when the puck bounces off a side board. The puck's y value is set to zero when it has contact with the ice so the puck does not bounce up and down on the ice.

For the purpose of creating a dynamic scene, randomness is introduced to the action as described in the following list. For comparison, the hockey rink is 200 units long with the nets placed at $x = -77$ and $x = 77$ units. The z direction is across the ice and the positive y direction is up. The velocity and position of players and the puck is updated every 17 ms (approximately 60 frames per second). In the simulation there is one time slice per frame. Thus the terms *time slice* and *time between frames* imply the same amount of time (i.e. 17 ms), although *time slice* refers to the physical simulation where as *time between frames* refers to the time before rendering the next image.

- Deciding which team moves first at each frame is randomized. Thus if the blue and red player reach the puck at the same time then there is a 50% chance for each player that they will control the puck at that time slice.

- The maximum velocity players reach is randomized between 0.9 and 1.15 units per time slice for each player. Similarly the maximum acceleration is randomized between 0.4 and 0.5625 units per time slice squared. Also, a player's acceleration is randomized at each time slide to add ± 0.02 units per time slice squared.

- When the puck is bumped forward by a player, it receives 1.6 times the player's velocity in the x direction and the player's velocity in the z direction $\pm 10\%$ (recall the x direction is lengthwise on the ice and the z direction is across the ice).

- When passing the puck, the range of the puck's velocity is 2.5 times the player's velocity ± 0.6 units per frame in the x direction and ± 0.4 units per frame in the z direction.

- When the puck is shot on net, its velocity is as follows: 4.0 ± 0.65 units per frame in the x direction, 0.7 ± 0.7 units per frame in the y direction and 4.0 ± 1.25 units per frame in the z direction.

- The goalie can hold the puck if it is up to 6 units away from the goalie's position. At each frame there is a 10% chance that the goalie will miss the puck.

The cumulative effect of the randomness introduced to the game is that the resulting action is unpredictable in that the puck and players' locations at any point in the game are only known at the start of the game, or after a goal. Thus the scene is a dynamic one.

4.2 Maze Game

The maze game is a third person perspective shooter where the human user controls an avatar to find target boxes in a 3D maze. The user can shoot a box at target boxes as shown in Figure 4.3. In Figure 4.3 the target box has a question mark texture, while the box fired by the player (a firing box) is to the left of the target box. After a target box has been hit the texture changes to a happy face. The user wins when all target boxes have been hit, either by a firing box or the avatar, as shown in Figure 4.4. 32 virtual cameras encircle the avatar in two rings parallel to the xz plane, consisting of 16 cameras in each ring, as depicted in Figure 4.5. The nearer ring is located at three units from the avatar, and the further ring is located at seven units from the avatar. To decrease the degree to which the avatar's head blocks the user's view, cameras to the left or right of the avatar are offset slightly left or right respectively (0.5 units for the outer ring and 0.35 units for the inner ring). Cameras in the outer ring are positioned at 0.5 units above the xz plane, whereas cameras in the inner ring are positioned at 0.9 units above the xz plane, the latter gives the perspective of looking over the avatar's shoulder. Smooth camera changes are performed by linearly interpolating between camera views in angle and position, rather than having abrupt camera feed changes, as in the hockey game.

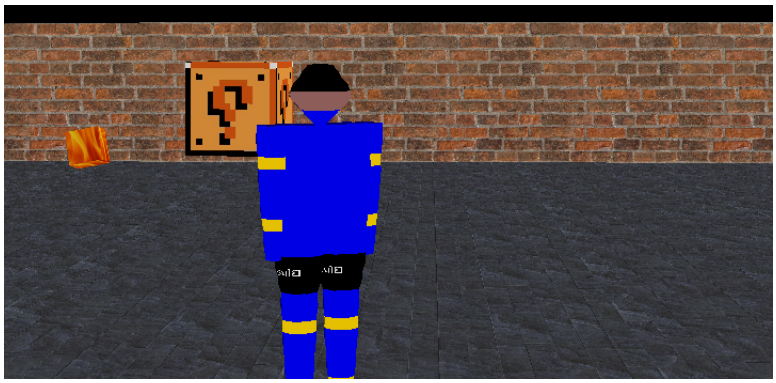

Figure 4.3: Maze Game - player firing a box (left) at a target box

In the maze game there is no randomness introduced by the program, but the avatar is controlled by a human user. Thus the user's actions are unknown to the system beforehand and, from the perspective of a designer, the scene is dynamic.

4.3 Chapter Summary

This chapter introduces the two test environments for the SCSP camera selection system. The hockey game is a spectator viewpoint game where the players are controlled by the computer.

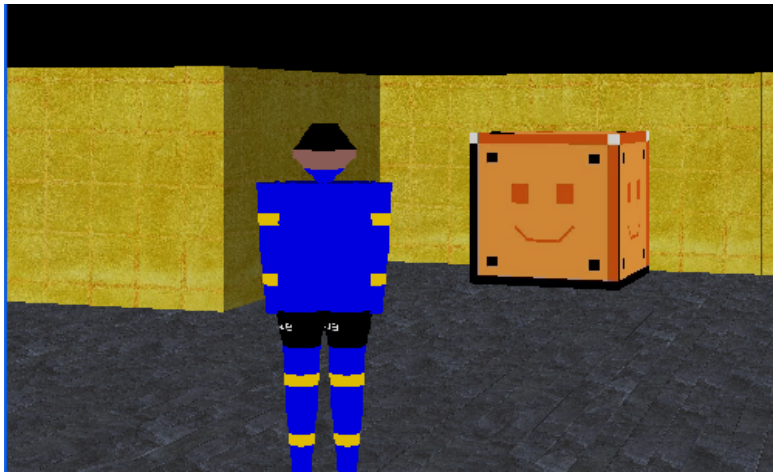

Figure 4.4: Maze Game - user has won

While the hockey players' logic is simple, the overall play is non-trivial. Randomness introduced into the players' actions creates a test system for dynamic scenes. By default, ten cameras are placed at fixed locations, but an arbitrary number of cameras in random locations can be specified at run time.

The maze game provides a third person perspective viewpoint. The user controls an avatar to find target boxes in a three-dimensional maze. Since the user's actions are not known beforehand, the game also produces dynamic scenes.

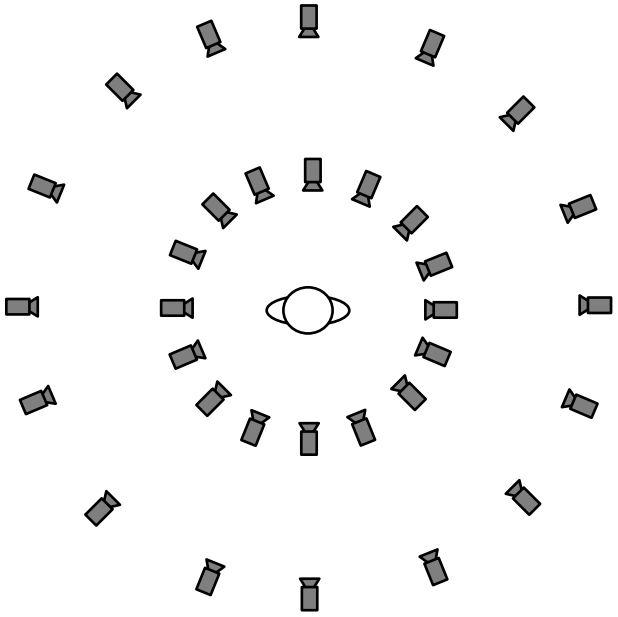

Figure 4.5: Camera Placement Around Avatar in Maze Game as depicted from a top down viewpoint.

Chapter 5

Design of the SCSP Camera Selection System

Current camera selection systems, such as the method presented by He *et al.*, perform camera selection, but with shortcomings. First, the system will typically specify transitions only to a limited number of cameras. This approach may work for scripted scenes where the action is known beforehand, but can be problematic in dynamic scenes where it is preferable to choose among many cameras at run time. For example, in a hockey or soccer game the next camera selected may be the one that best shows the puck or ball, regardless of the previous camera. In He *et al.*'s method the user has to provide the guards on transitions (the conditions to satisfy) for changing to another camera. The number of guards to specify is the sum of the number of potential cameras from each camera. For example, if there are ten cameras, and transitions are specified so as to transition to any other camera, then there are 90 guards to specify assuming that a guard is not required for a camera to transition to itself.

In addition to the designer potentially specifying many guards, current systems can remain with a suboptimal camera selection choice when no guard is fully satisfied. Suppose in a soccer game, camera three is currently selected, camera four has a partial view of the ball, and the ball is out of the field of view of all other cameras (including camera three). Also suppose that the designer specifies in the guards that the ball be visible in the camera's field of view. If the guard on the transition to camera four requires the ball to be fully visible, then the system will continue to select camera three even though camera four shows more of the soccer ball. A similar situation occurs when all cameras have the ball in their field of view. In this case all of the guards on transitions are satisfied and the system cannot differentiate between them (the ball may be more centered in one camera's field of view, which may be more preferred). If the guard is instead specified that the ball be in the center of the camera's field of view, then the initial problem will occur when the ball is in the field of view, but not centered. The camera selection system may remain with a suboptimal selection since the guard is not fully satisfied.

A problem with He *et al.*'s idiom approach is the hard constraint nature of the guards on the transitions. An additional problem is that each guard is designed independently of the other guards, resulting in the specification of many guards for the designer. Using soft constraints addresses the following shortcomings of He *et al.*'s idiom approach: choosing a camera feed from many cameras,

selecting a camera feed in a dynamic environment, and selecting the best camera feed when no camera feed satisfies all the specified requirements, or when more than one camera feed satisfies all the specified requirements. Using a SCSP approach means the solution is over finite, discrete variables from an arbitrary domain – arbitrary in the sense that it is up to the designer to choses which variables to include and the preferences to encode over those variables. Shown in Tables 5.1 and 5.2 are a summary of the constraints used in the hockey and maze games, introduced in Sections 4.1 and 4.2. These constraints will be explained in more detail in the remainder of this chapter. Specified soft constraints, introduced in Section 2.1.4, define the problem to optimize. Since constraints are defined under a SCSP interpretation, partial satisfaction of constraints is permitted.

Name	Use
KeepCentered	Prefer views where the puck is in the center of the camera's field of view
DistanceToCamera	Prefer views where the puck is closer to the camera
FrameCoherence	Prefer that the camera does not change too often or remain on one camera for too long
$WasFeed_x$	Prefer that the same camera feed remain on the same display
CenterScorer	Prefer that the scoring player appear in the center of the camera's field of view
SeeScorer	Prefer that the scoring player appear closer to the camera

Table 5.1: Summary of Constraints used in the Hockey Game

Name	Use
BiasCameras	Prefer camera viewpoints that are behind the player
SeeTarget	Prefer camera views where a target box is visible
SeeAvatar	Prefer camera views where the user's avatar is visible
SeeFiringBox	Prefer camera views where the firing box is visible
PassThroughWalls	Prefer to change to a camera that does not require passing through a wall

Table 5.2: Summary of Constraints used in the Maze Game

5.1 Selecting a SCSP Constraint Framework for Camera Selection

There are several ways to choose operators, but some may be more appropriate than others. A Fuzzy SCSP, briefly described in Section 2.1.4, could be used to represent the camera selection problem, but may not be sufficiently discriminating to accurately reflect a designer's preference. The issue lies in using *min* for combining tuples which can cause ties for the highest global preference, even when the input tuples are not equal. As an example, consider joining two independent constraints where $tuple_1$ and $tuple_2$ from the first constraint are to be combined with $tuple_3$ from the second constraint. Suppose that $tuple_1$ has a significantly higher preference than $tuple_2$, and that $tuple_3$ has a lower preference than either $tuple_1$ or $tuple_2$. When joined using the Fuzzy join, $tuple_1 \times tuple_3$ has the same preference as $tuple_2 \times tuple_3$ since the minimum in each case is $tuple_3$. Using a different join operation could rank $tuple_1 \times tuple_3$ as more preferred than $tuple_2 \times tuple_3$, which may be desirable.

A definition and use of a different semiring can avoid combined preferences from becoming equal when the input valuations are not equal. Consider,

$$S \quad = \quad \langle [0,1], f_+, f_\times, \mathbf{0}, \mathbf{1} \rangle$$

and $\mathbf{X}$, $\mathbf{C}$, and $\mathbf{D}$ remain as before. The operator, $f_\times$, is the usual arithmetic multiplication and can combine valuations similar to the Fuzzy join. The preference value of each combined tuple is defined using usual arithmetic multiplication:

$$C_{join}\left(x_1, \ldots x_{i_{k_i}}, x_{j_1}, \ldots x_{j_{k_j}}\right) \quad = \quad C_i(x_{i_1}, \ldots, x_{i_{k_i}}) \times C_j(x_{j_1}, \ldots, x_{j_{k_j}})$$

The operator f_+ is defined to be *max*.

Indeed, S is a semiring since it satisfies the properties of a semiring. Satisfying the properties of the semiring trivially follows from the properties of arithmetic multiplication (associative, distributive, etc.), and the properties of *max* (commutative, associative, etc.) [12]. For further details see Appendix B.

Returning to the meal example from Section 2.1.4, Table 5.3 shows the resulting preference distribution for the meal example using multiplication to combine preferences. Note that this SCSP join results in an ordering that is similar to the Fuzzy SCSP join in Table 2.1, but has more distinctions between the tuples.

To employ constraint propagation in a SCSP (Section 2.1.4), it is necessary to define a projection operator for S [31, 30, 51]. A projection operator for S can be defined as $C_{i\Downarrow\mathbf{X}^*}$, where $\Downarrow$ denotes

Main Dish	Fruit	Drink	Preference
pasta	apple	water	$.9 \times .5 = .45$
pasta	apple	milk	$.2 \times .5 = .1$
pasta	pear	water	$.5 \times .5 = .25$
pasta	pear	milk	$.1 \times .5 = .05$
stir fry	apple	water	$.9 \times .8 = .72$
stir fry	apple	milk	$.2 \times .8 = .16$
stir fry	pear	water	$.5 \times .8 = .4$
stir fry	pear	milk	$.1 \times .8 = .08$

Table 5.3: Example of Combined Preferences

projection, $C_i \in C$ is a constraint, and $\mathbf{X}^* \subseteq \mathbf{X}$ denotes the subset of variables on which to project. The new constraint $C_{i\Downarrow\mathbf{X}^*}$, is defined over the variables:

$$vars(C_{i\Downarrow\mathbf{X}^*}) \quad = \quad vars(C_i) \cap \mathbf{X}^*$$

where $vars(C_{i\Downarrow\mathbf{X}^*})$ and $vars(C_i)$ denote the variables involved in constraints $C_{i\Downarrow\mathbf{X}^*}$ and C_i respectively, and $\cap$ denotes intersection. Let $X_{i,1}, \ldots, X_{i,k_i}$ be the variables in $vars(C_{i\Downarrow\mathbf{X}^*})$ and $Y_{j,1}, \ldots, Y_{j,l_j}$ be the variables in $vars(C_i)$ but not in $vars(C_{i\Downarrow\mathbf{X}^*})$.

For each tuple in the projected constraint, the preference value is calculated as follows:

$$C_{i\Downarrow\mathbf{X}^*(X_{i,1},\ldots,X_{i,k_i})} \quad = \quad \sum_{Y_{j,1},\ldots,Y_{j,l_j}} C_i(X_{i,1}, \ldots, X_{i,k_i}, Y_{j,1}, \ldots, Y_{j,l_j})$$

where $C_{i\Downarrow\mathbf{X}^*}$ is the projected constraint over $X_{i,1}, \ldots, X_{i,k_i}$ and not over $Y_{j,1}, \ldots, Y_{j,l_j}$ (the order of the variables is not important in the projection). The $\sum$ operator is the prefix form of f_+, which is the application of the f_+ to a set of tuples. Remember that multiple tuples in C_i project onto the same tuple in $C_{i\Downarrow\mathbf{X}^*}$, unless $vars(C_i) \subseteq X^*$. In this semiring implementation, $\sum$ returns the maximum preference value of the tuples considered, since f_+ is defined as max.

For example, projecting the preference function in Table 5.3 from the set of variables {Main Dish, Fruit, Drink} onto the set {Main Dish} is computed as follows, with the resulting preference displayed in Table 5.4.

$$\begin{aligned} max\,(.45, .1, .25, .05) &= .45 \\ max\,(.72, .16, .4, .08) &= .72 \end{aligned}$$

Main Dish	Preference
pasta	0.45
stir fry	0.72

Table 5.4: $C_{Table\ 5.3\Downarrow\{\text{Main Dish}\}}(\text{Main Dish})$

A SCSP solver joins constraints to construct the global preference, and the resulting tuple with the highest preference is selected as the solution. In the meal example, shown in Table 5.3, the tuple with highest preference determines the choice of foods for a meal. In the case of a camera selection system, the tuple with the highest preference determines the camera feed to select.

An advantage to this SCSP approach is that constraints are defined modularly, based on inferred independence of a human designer, resulting in fewer tuples for the designer to specify than specifying the global preference directly. The case is similar to the argument used by Pearl when discussing independence in probabilistic relationships. Pearl claims people easily recognize independence relationships, compared to providing numerical estimates, when discussing probabilistic knowledge:

> Further light on the structure of probabilistic knowledge can be shed by observing how people handle the notion of independence. Whereas a person may show reluctance to giving a numerical estimate for a conditional probability $P(x_i|x_j)$, that person can usually state with ease whether x_i and x_j are dependent or independent, namely, whether or not knowing the truth of x_i will alter the belief in x_j. [70]

Recognizing independence relationships reduces the number of preferences the designer must specify (specify the preference of multiple small tables rather than provide the global preference directly), and recognizing independence relationships is easier than providing numerical parameters. This style of preference encoding accommodates arbitrarily complex relationships, while permitting simplifications where independence relationships hold.

5.2 Variables used in SCSP Constraints for Camera Selection

In the SCSP approach presented in this thesis, soft constraints are classified into two main types: static constraints set at design time, and dynamic constraints set at run time. Constraints set at design time represent the preferences of a designer, for example, that a subject appear in the center of the screen. Constraints defined at run time reflect the current environment, for example, that a subject is currently out of the camera's view. Without the information encoded by dynamic constraints, the solver may choose an optimal solution that reflects fantasy rather than reality. For

example, the solver may select a solution with the subject on screen and centered, where in reality, the subject is out of the camera's view.

Consider an example where the designer prefers the puck in the center of the camera's field of view. This preference, named *KeepCentered* can be encoded in a table, as shown in Table 5.5. Here $\mathbf{X} = \{Location\}$, $\mathbf{D} = \{\{center, border, out\text{-}of\text{-}view\}\}$, and $\mathbf{C} = \{KeepCentered\}$.

Location	Preference
center	1.0
border	0.7
out-of-view	0.3

Table 5.5: Static *KeepCentered* Preference

Note that the preference values lie between 0 and 1, where a higher preference value implies more preferred. Currently the SCSP solver would find a highest preference of 1.0 for $\{Location = center\}$.

The method, as described so far, selects the tuple from the global preference with the best camera feed, given the model of the problem as represented under a SCSP interpretation. The tuple selected, however, does not indicate which camera feed caused that tuple to be chosen, but merely contains the values of the variables the designer chose to represent the problem, and the preference. To address this issue of not knowing which camera feed to select, a variable *feed* indicates which camera feed is responsible for a tuple having the best preference. The *feed* variable indicates which camera feed to select, and can be retrieved from the tuple with the highest preference to indicate which camera feed will be selected. Returning to the preference shown in Table 5.5, suppose the example system has two equally preferred camera feeds, as shown in Table 5.6.

Feed	Preference
one	1.0
two	1.0

Table 5.6: *CameraFeed* Preference

Now the tuples with the highest preference are $\{Location = center, Feed = one\}$ and $\{Location = center, Feed = two\}$. Suppose that the hockey puck is currently in the border region of camera one's field of view and out of camera two's field of view. These constraints can be represented by a dynamic constraint, as shown in Table 5.7, set by the system during execution.

The tuple with the highest preference is now $\{Location = border, Feed = one\}$ and the preference

Feed	*Location*	Preference
one	*center*	0.0
one	*border*	1.0
one	*out-of-view*	0.0
two	*center*	0.0
two	*border*	0.0
two	*out-of-view*	1.0

Table 5.7: Dynamic *KeepCentered* Preference

value is 0.7. The *Feed* variable indicates that camera feed one should be displayed on the screen.

5.3 Multiple Displays

The SCSP selection system, presented so far, applies to one display but extends to multiple displays under the existing framework. When using multiple displays, the variables that represent the settings for the cameras are duplicated for each display to enable each display to have its own preference over camera variables.

Suppose in the hockey game a designer prefers the puck be centered in the first display, but out of view on the second display. A preference to not show an item or character may seem strange as a preference for a hockey game, but may encode a designer's preference to show events happening away from the puck, such as pulling the goalie[1]. In other genres of game, the designer may want to introduce a sense of mystery or suspense by delaying showing an enemy boss. One method of specifying a preference to show different areas of interest on different displays is to use the *Location* variable, and set a high preference to show the object on one display, and a low preference on the other display. However, if the same *Location* variable is used on each display, selecting a near zoom on one display also selects a near zoom on the other displays.

Duplicating the variable name with an index for each display, for each non-feed variable, provides a unique variable name for each display. Table 5.8 shows two *KeepCentered* preferences for two displays to encode the designer's preference to show the puck only on one screen. A corresponding dynamic constraint for each preference synchronizes the SCSP selection with the virtual environment. When a variable state is shared among all displays, such as a goal being scored in the hockey game, all displays can use the same variable. In this case there is no need to duplicate the variable name since synchronizing this particular variable's state for all displays is the desired

[1]This examples serves as possible motivation for this preference. Pulling the goalie is not implemented in the simulator.

behaviour.

$Location_1$	Preference	$Location_2$	Preference
center	1.0	*center*	0.3
border	0.7	*border*	0.7
out-of-view	0.3	*out-of-view*	1.0

Table 5.8: Static *KeepCentered* Preferences with Two Displays

5.4 Additional Preferences for the Hockey Game

Section 5.2 introduced the *KeepCentered* constraint to keep the puck in the center of a camera's field of view. Another preference, *DistanceToCamera* influences how large the puck appears on the display, as shown in Table 5.9[2]. The ratios between the preferences are not as large as with *KeepCentered* shown in Table 5.5, implying that in an over-constrained problem *KeepCentered* will be satisfied before *DistanceToCamera*. The corresponding dynamic constraint for *DistanceToCamera* is not shown but is similar to Table 5.7.

Distance	Preference
near	1.0
medium	0.8
far	0.5

Table 5.9: Static *DistanceToCamera* Preference

The designer may want to influence how long a display remains on the same camera feed. A temporal constraint, *FrameCoherence*[3], prevents rapid changes between camera feeds or remaining on one camera too long. As Table 5.10 shows, feeds currently used for less than five seconds are preferred to camera feeds selected for more than ten seconds. The last two values in Table 5.10 are values that can only be used to describe cameras that are not currently selected.

A potentially unwanted viewing effect occurs when using *FrameCoherence* and multiple displays. A camera feed selected for less than two seconds has a high preference, but the preference does not

[2]The valuations are arbitrarily determined by a user. These valuations may seem high, but in this case, were choosen to reflect that all settings of *Distance* are acceptable, although some are more preferable. This is in contrast to the *KeepCentered* constraint, when in some cases, *out-of-view* is usually unacceptable.

[3]We call this constraint *FrameCoherence* since its function is to maintain camera frame coherence, similar to the frame coherence objective in [47].

Duration	*Pref*
selected for less than two seconds	1.0
selected two to five seconds	0.9
selected five to ten seconds	0.4
selected more than ten seconds	0.3
selected less than two seconds ago	0.1
not recently selected	0.7

Table 5.10: Preference for Frame Coherence

specify on which display the feed should be shown. Thus, the feed could rapidly switch between different feeds. To avoid this, a $wasFeed_x$ variable is introduced for each display, where x indicates the display. To prefer the same feed remain on a given display, the designer specifies the preference of selecting a feed based on which feed is currently selected on a given display. When used with fixed position cameras, $wasFeed_x$ can also be used to design a constraint to avoid switching to another feed that is less than 30° different from the current camera, a desirable property with jump cuts [46]. Dynamic constraints using $wasFeed_x$ differ from dynamic constraints presented so far, in that the previous dynamic constraints are binary (involving two variables) where dynamic constraints using $wasFeed_x$ are unary (over one variable).

The designer may decide to change preferences depending on game events, such as a goal being scored in the hockey game. When combined with a dynamic *GoalScored* constraint, a *goalScored* variable with domain {*yes*, *no*} allows the designer to specify preferences depending on whether a goal is scored or not. For example, the preference from Table 5.5 can be replaced with the preference shown in Table 5.11 to indicate that the puck is preferred in the center of the field of view only when a goal has not been scored. Note that equal preference values indicate that the designer is indifferent between the choices.

The designer may choose to see the scoring player after the goal has been scored. The preferences *seeScorer* and *centerScorer* are similar to the preference in Table 5.11, but with respect to the scoring player rather than the puck.

While the constraints presented so far may appear simple on their own, the combination of preferences used is non-trivial. A more complicated constraint graph is shown in Figure 5.1 using 4 screens, 29 variables (shown in boxes), and 59 constraints (shown with lines) . Constraints ending in a dot imply unary constraints. Static constraints are shown in black while dynamic constraints are shown in red. Recall that the numbers on the ends of variables indicate settings for the indicated feed, these numbers are used to provide a unique variable name for each display. Notice the constraint graph shows static constraints between camera feeds. These constraints encode the

goalScored	*Location*	Preference
yes	*center*	1.0
yes	*border*	1.0
yes	*out-of-view*	1.0
no	*center*	1.0
no	*border*	0.7
no	*out-of-view*	0.3

Table 5.11: Static *KeepCentered* preference when considering if a goal has been scored or not.

designer's preference that each display show a different camera viewpoint. Tuples that select the same camera for two different feeds are assigned a preference of 0.1, while other tuples are assigned a preference of 1.0. The implentation of preferences shown in Figure 5.1 is a proof of concept, not a proposed camera system. Commercial interests could use this system to encode more complex preferences using the SCSP approach.

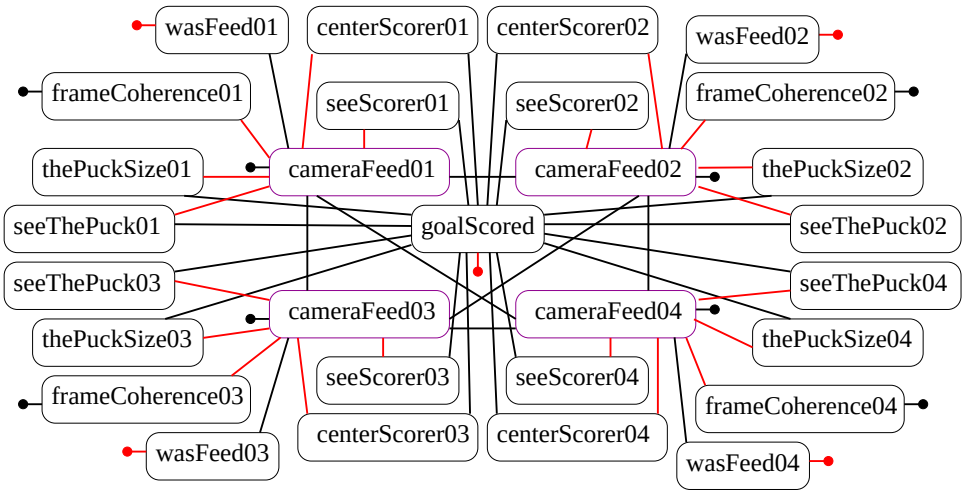

Figure 5.1: Constraint Graph Example using Four Displays

5.5 Preferences for the Maze Game

One way the maze game differs from the hockey game is that the user controls the avatar. Consequently, views behind the avatar are preferred since the user's view and the avatar's view are aligned and make the avatar easier to control. The constraint *BiasCameras* prefers view points behind the avatar. Shown in Table 5.12, *BiasCameras* is similar to Table 5.6 but has 32 domain

values for *Feed*. Notice that camera feeds *1*, *2*, *3*, *16*, *17*, and *18* have a higher preference value than other tuples, since these camera feeds are behind the avatar. Also notice that camera feeds from the outer ring (1 through 16) are preferred to those on the inner ring (17 through 32)[4]. The static constraint *BiasCameras* does not need a corresponding dynamic constraint, as the cameras are fixed relative to the user's avatar. Consequently, the number of the camera behind the avatar does not change during the maze game.

Feed	Preference	*Feed*	Preference
1	1.0	*17*	0.8
2	0.9	*18*	0.7
3	0.8	*19*	0.6
4	0.7	*20*	0.5
5	0.6	*21*	0.4
6	0.6	*22*	0.4
7	0.6	*23*	0.4
8	0.6	*24*	0.4
9	0.6	*25*	0.4
10	0.6	*26*	0.4
11	0.6	*27*	0.4
12	0.6	*28*	0.4
13	0.6	*31*	0.4
14	0.6	*30*	0.4
15	0.7	*31*	0.5
16	0.8	*32*	0.6

Table 5.12: *BiasCameras* preference

Occlusion constraints keep particular characters or objects in view. In the maze game the user searches for targets, so views with a target are preferred to those without. However, the target may be occluded by a wall. A constraint, *SeeTarget* over the variable *targetVisible* encodes the preference to see a target, as shown in Table 5.13. The preference to see the avatar, *SeeAvatar*, and to see a firing box, *SeeFiringBox*, are similar, and shown in Tables 5.14 and 5.15 respectively. In this example, *SeeFiringBox* reflects an equal preference to see the firing box well, or some, and a low preference for a poor view of the firing box.

Users may find a camera change that passes through a wall disorienting. The preference *SeeA-*

[4]See Figure 4.5 for camera locations for this example.

targetVisible	Preference
unoccluded	1.0
partially occluded	0.9
occluded	0.4

Table 5.13: *SeeTarget* preference

seeAvatar	Preference
unoccluded	1.0
partially occluded	0.6
occluded	0.1

Table 5.14: *SeeAvatar* preference

vatar decreases the preference for views where the avatar is not visible, but does not affect the path between the cameras, which may pass through a wall. The preference *PassThroughWall* over the variable *passThroughWall* decreases the preference for camera transitions that pass through a wall. Details of the implementation of the corresponding dynamic constraint are presented in Chapter 6.

5.6 Chapter Summary

This chapter introduces the constraints used under the SCSP camera selection system as implemented in the hockey and maze games. The join operator, $\otimes$, is implemented using arithmetic multiplication for each combination of tuples. Preferences for the hockey game are already summarized at the beginning of this chapter in Table 5.1, while the maze game preferences are summarized in Table 5.2. With multiple displays, additional static constraints may require each display to show a different camera feed.

seeFiringBox	Preference
unoccluded	1.0
partially occluded	1.0
occluded	0.1

Table 5.15: *SeeFiringBox* preference

passThroughWall	Preference
no	1.0
few	0.2
lots	0.1

Table 5.16: *PassThroughWall* preference

Chapter 6

Implementation Details

This chapter presents the implementation of several dynamic constraints in the hockey and maze games, and explains details of the SCSP solver. Dynamic constraints are set by the system at run time; this chapter explains how they are set. Chapter 5 introduces static and dynamic constraints, and a high level view of combining constraints. An exhaustive join, however, is intractable for large problems, so a branch and bound search process is used instead. While a branch and bound search is also intractable for large problems its execution time may be much less than an exhaustive join due to the effect of pruning branches. The times required to perform a branch and bound search on example domains are presented in Chapter 7.

6.1 Frame Rate in Simulation

The amount of time the SCSP solver has to select a camera depends on how frequently the camera selection is made. While the camera selection procedure could be run every frame of the simulation, it may not be desirable as it could result in overly frequent camera changes. The timers in the hockey and maze games are set to update objects' movement at 60 frames per second. An acceptable viewer experience was found when the hockey game solves for a camera selection every 30 frames and the maze game every 20 frames.

6.2 Dynamic Constraints

Dynamic constraints require the system to set the preference values at run time to reflect the current situation in the virtual environment. Each dynamic constraint has a function in the virtual environment that returns an array of preference values appropriate to the constraint. For example, the dynamic constraint for *KeepCentered*, shown in Table 5.7, returns an array of preferences ordered by *Feed* and then *Location*. Thus, the function determines the location of the puck in the field of view for each camera. To determine the location of the puck on the display, the puck is projected to the display using the matrices, described in Sections 2.3.1 and 2.3.2, which are retreived from the OpenGL pipeline. The puck's x and y coordinates on the display are returned as floating point values in which values of -1 and +1 are at the limits of the display. If the puck's x and y values are

between -0.4 and +0.4 then the puck's location on the screen is *center*. Values outside this range, but on the display map the puck location to *border*, and values that result in the puck not shown on the screen map to *out-of-view*. The *DistanceToCamera* is similarly determined, but uses the puck's z coordinate. The dynamic constraints for *CenterScorer* and *SeeScorer* are similarly encoded but use the scoring player as input instead of the puck. When no goal has been scored these values are set to *out-of-view* and *far* respectively.

The preferences for the binary dynamic constraint, *FrameCoherence*, are set using an array of times and an array of cameras currently selected. The array of times records the last time each camera feed was selected. The array of cameras records which camera is selected on each display. Each time the system selects a camera these arrays are updated. The preference values of each tuple are set to 1.0 or 0.0 by comparing the current time to the last time the camera was selected. For example, if the same camera is selected, and has been selected for less than two seconds, then the state *selected for less than two seconds* would receive a preference value of 1.0 while the other states would be assigned preference values of 0.0. Similarly, the unary dynamic constraint for $wasFeed_x$ is set by examining which camera is currently used for display x.

The unary dynamic constraint for *GoalScored* is dependent on whether a goal is scored or not. A goal is considered to be scored for 200 frames after the puck crosses the goal line, as this is roughly the time required for players to return to a face-off position.

To set the dynamic occlusion constraints in the maze game the avatar and each object are bounded by non-visible boxes, which are divided into smaller boxes and projected to an imaginary 2D screen, defining rectangles. The screen is imaginary in the sense that the bounding boxes are not displayed to the user. The imaginary screen, however, is aligned with the user's display so that decisions performed using the imaginary screen appropriately apply to the game.

Target and firing boxes are divided into 8 smaller boxes, as shown in Figure 6.1, while the avatar is divided into 27 smaller boxes ($3 \times 3 \times 3$). Other objects in the maze, such as wall segments, are similarly projected to rectangles on an imaginary screen (but not first subdivided) and compared to the rectangles of the object of interest to determine overlap.

The dynamic constraint, *TargetVisible*, is constructed using a simple counting technique. Since each smaller box projected to the imaginary screen defines a rectangle, the number of resulting non-overlapped rectangles determines the dynamic constraint assignment to *TargetVisible*. Since an object can't be occluded by an object behind it, a rectangle from a smaller box is only considered occluded if it is overlapped by a rectangle from a closer occluding object. When setting the dynamic *SeeAvatar* constraint, the system assigns a preferences of 1.0 to *unoccluded* if 15 or more of 27 rectangles are visible, *partially occluded* if between 10 and 14 rectangles are visible, and *occluded* if fewer than 10 rectangles are visible[1]. Preference values of 0.0 are assigned to the other states.

[1]This is one implementation. Other groupings for occlusion are possible.

Firing and target boxes are similarly assigned preferences to *SeeFiringBox* and *SeeTarget* based on more than six, three to six, or fewer than three visible rectangles respectively.

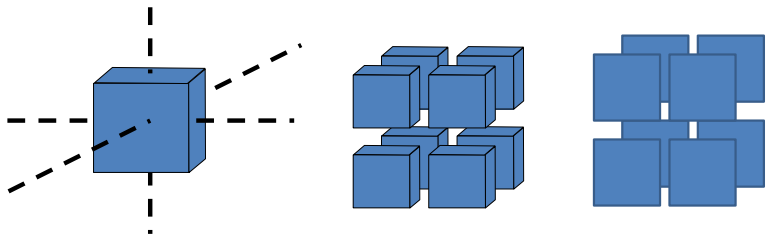

Figure 6.1: Occlusion constraints are determined by counting the number of visible rectangles. An object is first bound by an imaginary box (left). The box is divided into smaller boxes (middle). The smaller boxes are projected to rectangles onto a screen, for counting which rectangles are visible (right).

The dynamic constraint values from *SeeAvatar* are used to set the dynamic constraint *PassThroughWall*. The system considers potential paths, which are a sequence of cameras between a starting camera and ending camera. Since camera changes interpolate the angle between the currently selected camera and the next selected camera, the interpolated camera values periodically align with camera values available for selecting, for which occlusion information is known from setting the *SeeAvatar* dynamic constraint. To determine if a path is occluded, the system counts the number of occluded cameras along a proposed path of the rings of cameras as shown in Figure 4.5. When changing from one camera to another, the camera rotates left or right by determining which angle is smaller between the current and next camera position. Note these cameras may not be adjacent to one another in position. If passing along the inner or outer ring, only cameras on that ring are considered. If passing from one ring to another, occluded cameras from both rings are considered.

If no cameras along the proposed path are occluded, *PassThroughWall* assigns a preference value of 1.0 to state *no* and 0.0 to states *few* and *lots*. Similarly, if one camera is occluded then 1.0 is assigned to *few* and a preference value of 1.0 is assigned to state *lots* if two or more cameras on the proposed path are occluded (with other states assigned a preference value of 0.0).

6.3 SCSP Solver

The global tuple with the highest preference is found via a best first, branch and bound tree search. The search assigns single values to a variable at each node until, at leaf nodes, all variables are assigned a value representing an assignment. A maximum degree heuristic determines the order in which variables are assigned values; variables participating in more constraints are assigned values earlier than variables participating in fewer constraints [84, 58]. Since dynamic constraints prune many branches the memory used by the best first search solver is typically not large relative to the

amount of memory available in modern computer systems. The bound on the memory required is most strongly determined by the number of displays and secondly by the number of camera feeds. A more formal analysis appears in Section 6.4. In memory restricted implementations a depth-first search could be used[2]. The best first strategy, however, has the potential advantage over the depth-first search of expanding fewer nodes in the tree since the best solution may be reached earlier in the search.

Partial assignments have a preference level associated with them that can be used to prioritize which nodes are expanded before others. As an example, to assign a preference level to a partial assignment, consider binary constraints. Assume the variables are ordered such that variables with lower indices are assigned values before variables with higher indices. Assume that the variables $X_1 \ldots X_k$ are assigned values $x_i \ldots x_k$ respectively. The preference value associated with a particular variable, i, can be determined by multiplying all the constraints involving i and variables with an index lower than i.

$$Ub_i \quad = \quad \prod_{j=1}^{i-1} C_{ji}(x_j, x_i), i > 1$$

where each Ub_i groups constraints based on the maximum index of the variables involved, and $\prod$ implies arithmetic multiplication, since the variables in the constraint are assigned values x_j and x_i. The purpose of stopping the product at $i-1$ avoids double counting, such as including constraints C_{12} and C_{21}. If the designer has not provided a preference function for a C_{ji}, then a preference value of all ones can be assumed, as this leaves the valuation unchanged. In implementation the computation is avoided as it does not change the valuation.

Ub denotes an upper bound since the valuations monotonically non-increase as each constraint is applied. The preference assigned to the partial assignment is the product of the Ub_i valuations.

$$Ub \quad = \quad \prod_{i=2}^{k} Ub_i$$

Note that at leaf nodes, the upper bound is the same as the actual valuation assigned to the tuple, since the partial assignment has become a full assignment. Non-binary constraints, such as unary constraints, can be used to calculate Ub in a similar fashion.

For example, if there are three variables assigned values, $X_1 = x_1, X_2 = x_2,$ and $X_3 = x_3$, then the partial assignment can be determined as follows (k=3).

$$Ub_2 \quad = \quad \prod_{j=1}^{2-1} C_{ji}(x_j, x_i)$$

[2]The best first search can be converted to a depth-first search by changing the ranking function, in the priority queue, such that more recently added nodes are preferred to less recently added nodes.

$$
\begin{aligned}
&= C_{12}(x_1, x_2) \\
Ub_3 &= \prod_{j=1}^{3-1} C_{ji}(x_j, x_i) \\
&= C_{13}(x_1, x_3) \times C_{23}(x_2, x_3) \\
Ub &= \prod_{i=2}^{3} Ub_i \\
&= C_{12}(x_1, x_2) \times C_{13}(x_1, x_3) \times C_{23}(x_2, x_3)
\end{aligned}
$$

The valuation of the partial assignment depends on the values assigned to the variables and the valuations of the preference functions C_{ji}.

The best first search is implemented using a priority queue. Retrieving the next branch to expand from a priority queue, based on current preference value, expands branches with higher preference before branches with lower preference. The search begins pruning branches once a solution from a complete assignment has been found. If a branch to expand has a lower current preference than the assignment already found then the search discards the branch as it contains no solutions with higher preference than the solution already found.

Dynamic constraints cause aggressive pruning, as often only one branch contains a non-zero preference value. Consequently, propagation techniques to accelerate the search were not used. Camera selection systems with a large number of constraints may execute slowly due to a large number of nodes for the search process to expand. While they may cause aggressive pruning, dynamic constraints do not have the same pruning effect on output variables since it is not known beforehand which value assignment will result in the maximum preference level. As a result, mulitiple output variables, such as using the camera selection system with multiple displays, will result in an increase in the number of nodes to consider, assuming there is more than one camera to select. In such cases constraint propagation techniques may be added to the system, or the SCSP solver can be converted into an approximate native code solution, as discussed in Chapter 7.

6.4 Discussion on Complexity

The number of leaf nodes a general branch and bound searches is bounded by n^b, where n is the depth of the tree (i.e. the number of variables to set), and b is the branching factor (i.e. the largest domain of the variables). With the camera selection SCSP system, however, the combination of dynamic constraints with a best first value ordering leads to a possible tighter bound on the number of leaf nodes. As implemented, dynamic constraints will have only one non-zero value. As a result of using multiplication for combining tuples, the search process will traverse the branch with the non-zero value first due to the best first value ordering. When the search later begins traversing

a branch with a zero preference, the branch will be pruned, since another, non-zero solution will have already been identified.

The consequence on complexity analysis is, that during a typical search, only camera feed variables are expanded beyond a single branch while dynamic constraints effectively allow pruning of the majority of the sub-trees. Constraints affect the time required to select a camera, as they determine the preference of a tuple. The most significant influence on search complexity, however, is the number of displays (or output variables as mentioned in future work, Section 9.1), followed by the number of feeds. There are exactly d displays. For each display, f feeds are considered. Therefore the total number of choices to explore is f^d. All other variables in the model are set by dynamic constraints at run time and have a branching factor of one. Thus the worst case time complexity to select a feed for each display is $O(f^d)$. This result will be used in Chapter 7 to help transform experimental data into a form more suitable for computing an equation using linear regression.

Concerning memory use, in the worst case all of the $O(f^d)$ nodes of the tree are stored in the queue, waiting for expansion. Thus the memory use is bounded by $O(f^d)$[3].

6.5 Chapter Summary

Dynamic constraints require that the preference values be set at run time. Each dynamic constraint has a procedure for setting these preference values. For example, *KeepCentered* and *SeeScorer* project the puck or player to an imaginary screen to set the appropriate visibility. To set the *FrameCoherence* constraint, the last time each camera was selected is retrieved from an array. Occlusion in the maze game is handled by dividing and projecting a bounding box to the screen and counting the number of parts visible. The occlusion constraint is used in the *PassThroughWall* preference to determine if a camera transition passes through one or more walls.

The best tuple indicates which camera feeds to select and is found using a best first, branch and bound, tree search. When expanding the tree, variables involved in more constraints are expanded before those with fewer, and values resulting in a higher partial preference are explored before those with a lower preference. While the number of leaf nodes examined in a tree search is bounded in general by n^b, consideration of dynamic constraints allows a tighter bound of $O(f^d)$, where f is the number of feeds, and d is the number of displays.

[3]Recall from Section 6.3 that the memory use could be reduced, at the cost of a longer search time, by changing to a depth first search, rather than a best-first search.

Chapter 7

Acceleration Techniques

For larger SCSP problems, the process of searching for the tuple with the highest preference value can require an unacceptable length of time. If a camera selection is needed every frame at 60 frames a second, then camera selection must be complete in approximately 17 milliseconds, assuming the required processing resources are available to the SCSP selection algorithm for the entire 17 milliseconds[1]. Graphs in Section 7.1, which will be discussed further in this chapter, show some conditions under which camera selection is possible in less than 17 milliseconds, and some conditions where camera selection requires longer than 17 milliseconds. Propagation techniques, discussed in Section 2.2, can reduce the time required to find a solution but can still require an unacceptable length of time.

In a game using the SCSP camera selection system, such as the hockey simulation, configurations occur more than once. This means that the variables affecting the dynamic constraints have the same values as a previous search. Instead of searching for the tuple again, the previously computed camera feed can be retrieved from a cache (see Section 7.1) or a decision tree that has been compiled to native code (see Section 7.3). These methods return a choice of camera feed in less time, but the solutions may be approximate in the case of a decision tree, or a solution may not be found if it has not been entered into the cache. The cache is able to find more configurations of game states as the number of cache entries increases. The larger the number of samples used to build a decision tree, the more accurately it encodes the results of the SCSP search.

7.1 Cache

As time goes on in the simulation, configurations reoccur whenever dynamic constraints for the current evaluation are identical to a previous evaluation. Recall that the dynamic constraints specify the state of the simulation to the SCSP solver; a cache takes advantage of this temporal locality to reduce the average evaluation time to select a camera. Rather than repeat a search to find the tuple with the highest preference, the cameras to select can be retrieved from a cache using

[1]It may be that one core of the processor handles the camera selection system, while other cores handle other aspects of game pay .

a hash function.

The demonstration system requires more time to select from a higher number of cameras. Consequently, at some number of cameras the system can no longer be considered to function in real time. This limit will depend on the hardware used and the designer's definition of real time. This section presents graphs that show the limits of the system using a given set of hardware with, or without, using a cache. Graphs in this secton containing values that exceed a camera selection time longer than 16.6ms (60 frames per second) contain a dashed red line indicating the 60 frames per second limit. Simulations were performed on a Intel Core 2 Duo 2.4GHz computer with 8GB of RAM and Windows Vista 64 Professional (although the simulation executable is 32 bit). Two constraints, *KeepCentered* and *DistanceToCamera*, are found to result in acceptable camera selection quality so simulations with two, three, and four constraints were recorded[2]. During a simulation, the time needed to search for the camera feed to select was recorded twice when a camera selection was needed – once without a cache and once with a cache.

The simluation was allowed to execute for 500 samples, which corresponds to approximately 4.25 minutes of game play, long enough for both teams to score multiple goals. The actual time of the simulation may exceed 4.25 minutes, as the time required for camera selection is added to the game play time. The 4.25 minute duration is based on 500 samples, with a camera decision made every 30 frames and a 17 millisecond period between frames. Cameras are placed randomly (within a given range of locations and angles), and the cache is implemented using a hash table with arrays for items mapped to the same hash value. The time from each of the simulation's 500 samples were then averaged to determine an average time for camera feed selection.

To reduce the effect of outliers, the simulation was repeated five times and averaged, resulting in 2500 samples per data point on a graph. Outliers can occur due to the randomness in the simluation. For example, some of the abilities of the simulated hockey players, such as maximum speed, are randomized and it could be that one side scores many goals while the other scores none. This could result in more cache hits, since similar conditions occur when a goal is scored by a given team.

Figure 7.1 plots the average time to select a camera based on the number of cameras available. Figure 7.1 uses two constraints: *KeepCentered* and *DistanceToCamera*. The blue squares plot the average time without using cache to select a camera, while the orange triangles plot the time using cache.

As shown in Figure 7.1, the time to select a camera feed is approximately a linear function based on the number of cameras. Using linear regression, the equation to describe the time in

[2]Each simulation required approximately a month to complete, due to the number of samples, and adding a pause before performing a search. This pause provides the operating system time for background operations.

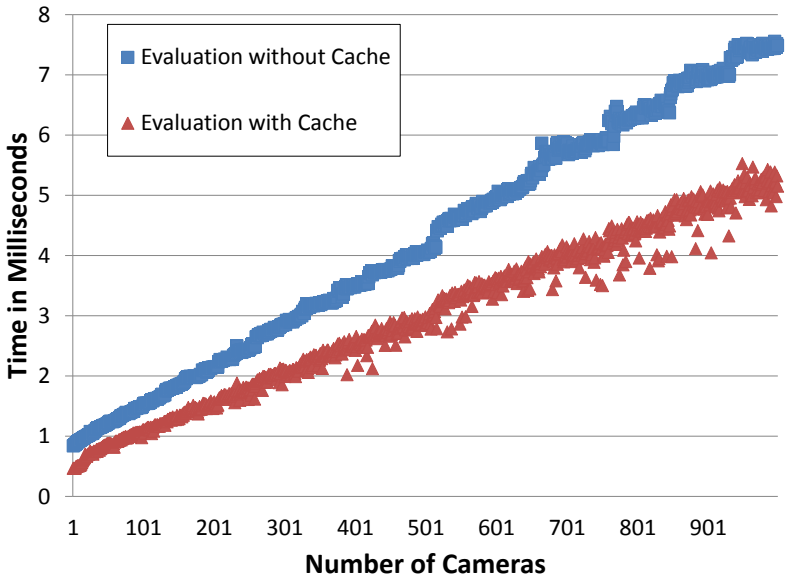

Figure 7.1: Computation Time: Cache vs. No Cache with One Display, constraints *Keep-Centered* and *DistanceToCamera*

milliseconds needed to select the camera feed (without using cache) is as follows:

$$\text{time for selection} \quad = \quad 0.006888 \times (\text{number of camera feeds}) + 0.8071$$

From this equation, the maximum number of cameras from which the camera can select a feed in $\frac{1}{60}^{th}$ of a second (i.e. 60 frames per second) is approximately 2302. When cache is used the maximum number of camera feeds to select among increases to 3388. This assumes that the SCSP system has dedicated resources, and does not have to wait for a CPU time slice or memory access. Consquently, 2302 describes an upper bound as the maximum number of cameras permitted per frame, which will decrease if other algorithms are using the same computational resources. Table 7.1 summarizes the computations for one, two, three and four displays with and without using cache. Tables 7.2 and 7.3 show values when *FrameCoherence* and *GoalScored* constraints are added. Note that using a cache decreases the average computational time, not each SCSP search.

Selecting a camera feed every frame may be unneccesary, as a designer may determine that acceptable camera selections can be done every 30 frames when the simulation is running at 60 frames per second. This decision results in a camera selection twice per second. If the SCSP camera selection system uses the half a second to select a camera, then the theoretical maximum number of cameras the system can select from increases to 72472 without using a cache and 105487 when using a cache. Values for two, three, and four displays are shown in the last column of Table 7.1[3].

[3]The calculation without cache and four displays resulted in a negative root for the theoretical number of cameras.

The first column of Table 7.1 provides the number of points on which the linear regression is based.

Cameras Tested	Displays	Cache used	Regression Equation	Maximum Cameras	Every 30 Frames
999	1	no	$y = 0.807100 + 0.006888x$	2302	72472
999	1	yes	$y = 0.623600 + 0.004734x$	3388	105487
100	2	no	$y = 2.800000 + 0.010760x^2$	35	214
100	2	yes	$y = 0.965200 + 0.006904x^2$	47	268
29	3	no	$y = 9.378195 + 0.000189x^3$	33	137
29	3	yes	$y = -2.168223 + 0.006645x^3$	14	42
26	4	no	$y = 124.400000 + 0.030440x^4$	-	10
26	4	yes	$y = -69.700000 + 0.013220x^4$	8	14

Table 7.1: Maximum Number of Cameras using constraints *KeepCentered* and *DistanceToCamera*. The x variable is the number of cameras and y is time in milliseconds. Maximum Cameras is for camera selection every frame, at 60 frames per second. Every 30 Frames applies to selecting a camera at 30 frame intervals, when running at 60 frames per second (i.e. camera selection every half a second).

Recall from Section 6.4 that the time required by the SCSP search is bounded by $O(f^d)$. Consequently, to determine a regression equation the times for two, three, and four displays are first transformed by using a model based on the square, cube, or fourth root, respectively, of the measured calculation time[4]. This exponential relationship is also reflected in the exponents in Tables 7.1 through 7.3. Appendix C provides graphs of the transformed data that visually verifies the linear relationship. The R^2 values for the regression equations in Tables 7.1 through 7.3 are all above 0.96 and all but four are above 0.99. The occasional negative y-intercept for the cache formulas in the regression equations may be explained since the equations are constructed assuming that the data is linear, when the beginning entries of data sets using cache do not appear linear until after approximately ten cameras. The graphs and tables show that the number of camera feeds from which the system can select is most strongly determined by the number of displays and then the number of constraints.

The likelihood of getting a cache hit increases as the number of entries in the cache increases. Consequently, when using a cache the average time for camera selection will decrease as the number of cache entries increases. Figures 7.13 through 7.16 repeat the experiments with four constraints but use 5000 samples per simulation instead of 500. For comparison, the results from the simulations

This is likely due to the relatively large intercept calculated by R.

[4]The regression equation was determined using the statistical software package R. R version 2.14.0 (2011-10-31) (C) 2011 The R Foundation for Statistical Computing.

Cameras Tested	Displays	Cache used	Regression Equation	Maximum Cameras	Every 30 Frames
999	1	no	$y = 0.914000 + 0.010270x$	1533	48596
999	1	yes	$y = 0.751100 + 0.008690x$	1831	57450
100	2	no	$y = 9.598782 + 0.000243x^2$	170	1421
100	2	yes	$y = 7.624495 + 0.017294x^2$	22	168
44	3	no	$y = -20.180000 + 0.044170x^3$	9	22
44	3	yes	$y = -47.260000 + 0.004326x^3$	24	50
21	4	no	$y = 220.2000000.058680x^4$	-	8
21	4	yes	$y = -56.5900000.054470x^4$	6	10

Table 7.2: Maximum Number of Cameras using constraints *KeepCentered*, *DistanceTo-Camera*, and *FrameCoherence*.

using 500 samples are also shown on the graph. The purple circles show the improvement afforded from having a larger number of entries in the cache.

Figures 7.1 through 7.16 show that the cache improves the average camera selection time but has noticeable lag with a large number of displays and cameras. The cache technique, however, may still be practical since best first search is an anytime algorithm. The search can return the best current feed selection in a timely fashion while a background process continues to search for the optimal feed. Subsequent requests, using the same dynamic constraints, use the improved camera selection from the cache. An ever increasing backlog of requests may be avoided by prioritizing background searches based on the frequency of access of the cache entry. Infrequent combinations of dynamic constraints may never be searched if the cache entry is eventually removed from the cache due to cache conflicts from other, more frequently accessed, cache entries.

Cameras Tested	Displays	Cache used	Regression Equation	Maximum Cameras	Every 30 Frames
999	1	no	$y = 0.858200 + 0.019840x$	796	25158
999	1	yes	$y = 0.662500 + 0.018090x$	884	27602
100	2	no	$y = 7.776151 + 0.048449x^2$	13	100
100	2	yes	$y = 5.110177 + 0.046619x^2$	15	103
33	3	no	$y = -4.962245 + 0.098270x^3$	6	17
33	3	yes	$y = -42.500000 + 0.094780x^3$	8	17
20	4	no	$y = -13.540000 + 0.146100x^4$	3	7
20	4	yes	$y = -389.900000 + 0.132900x^4$	7	9

Table 7.3: Maximum Number of Cameras using using constraints *KeepCentered*, *DistanceToCamera*, *FrameCoherence*, and *GoalScored*.

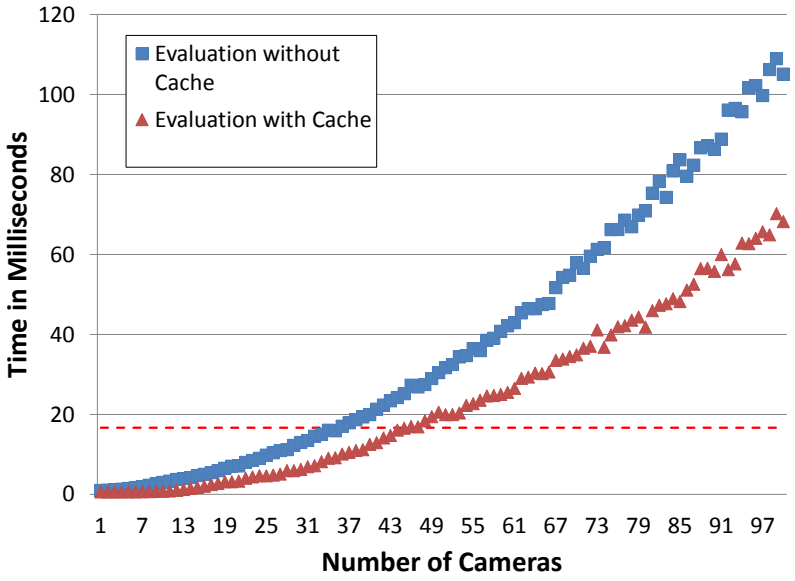

Figure 7.2: Computation Time: Cache vs. No Cache with Two Displays, constraints *KeepCentered* and *DistanceToCamera*

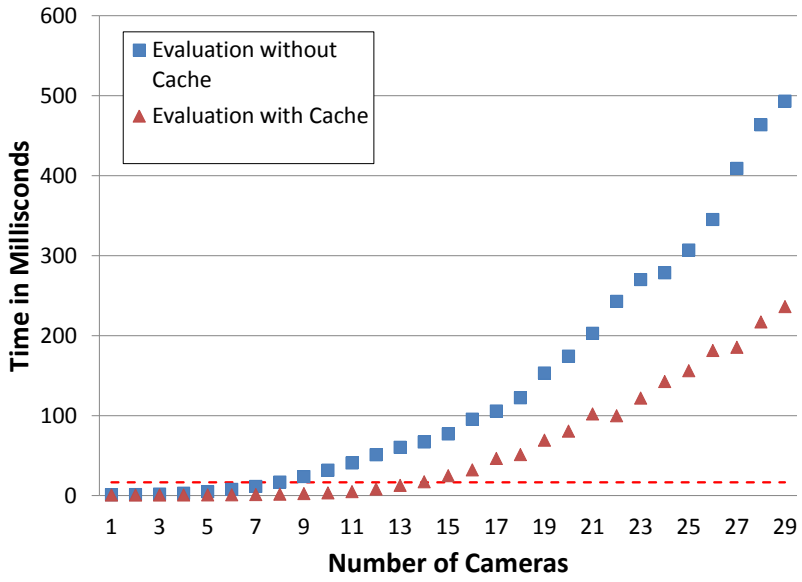

Figure 7.3: Computation Time: Cache vs. No Cache with Three Displays, constraints *KeepCentered* and *DistanceToCamera*

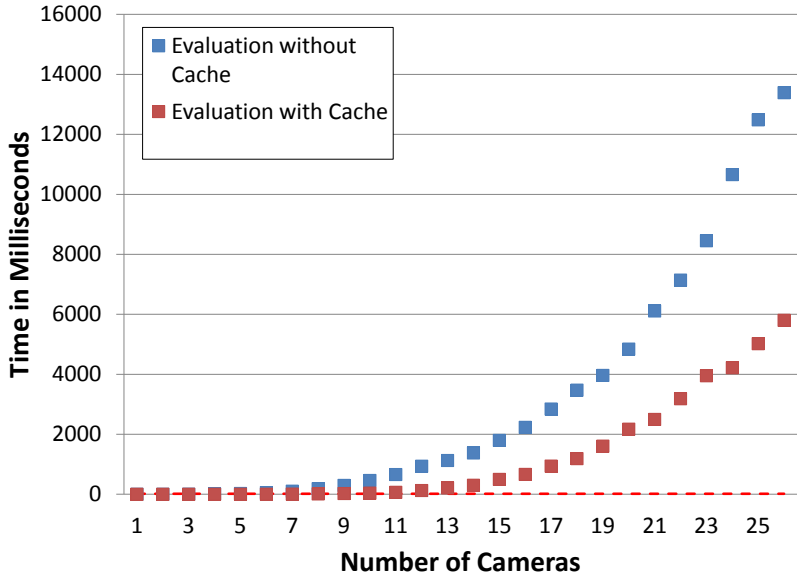

Figure 7.4: Computation Time: Cache vs. No Cache with Four Displays, constraints *KeepCentered* and *DistanceToCamera*

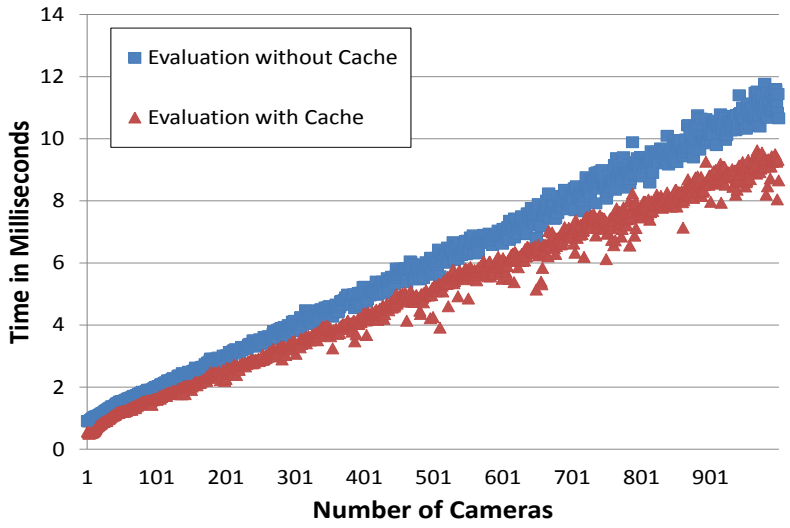

Figure 7.5: Cache vs. No Cache with One Display, constraints *KeepCentered*, *DistanceToCamera*, and *FrameCoherence*

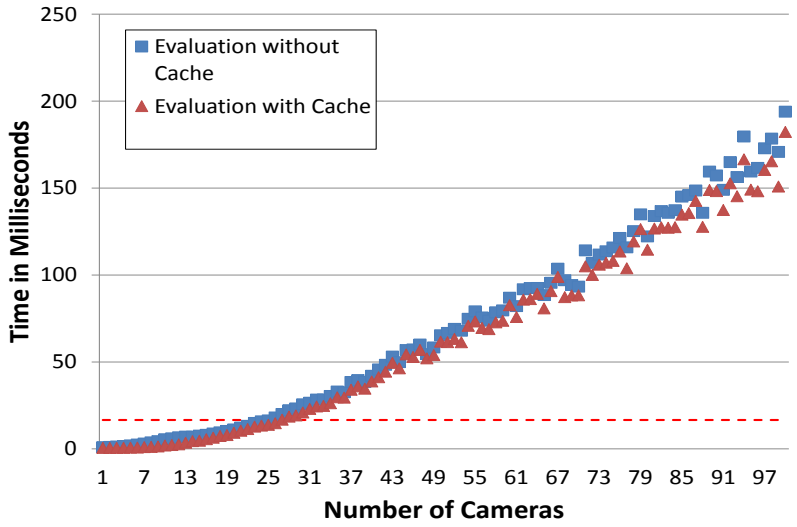

Figure 7.6: Cache vs. No Cache with Two Displays, constraints *KeepCentered*, *DistanceToCamera*, and *FrameCoherence*

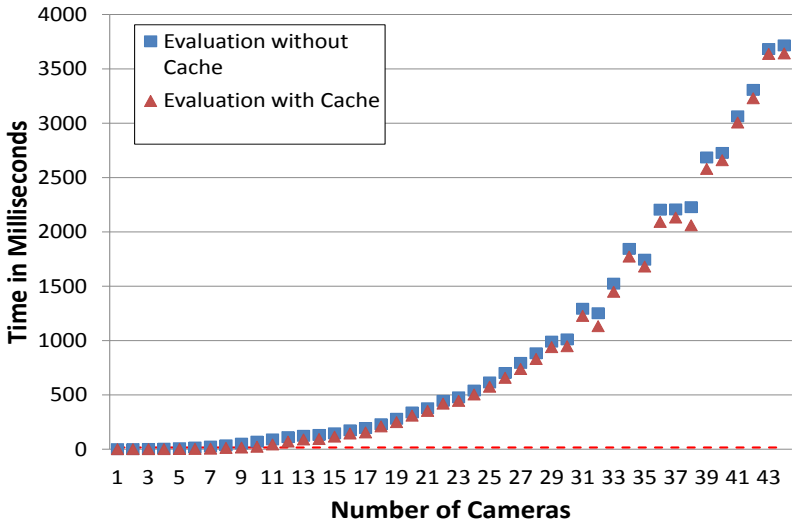

Figure 7.7: Cache vs. No Cache with Three Displays, constraints *KeepCentered*, *Distance-ToCamera*, and *FrameCoherence*

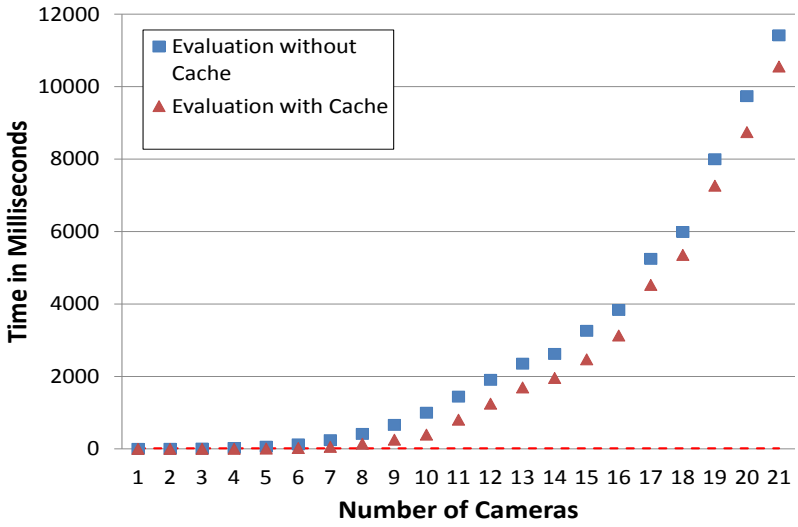

Figure 7.8: Cache vs. No Cache with Four Displays, constraints *KeepCentered*, *Distance-ToCamera*, and *FrameCoherence*

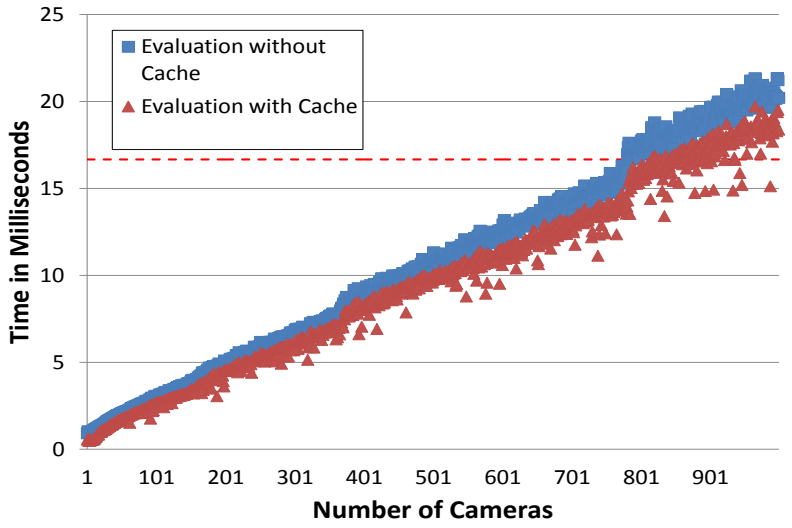

Figure 7.9: Cache vs. No Cache with One Display, constraints *KeepCentered*, *DistanceToCamera*, *FrameCoherence*, and *GoalScored*

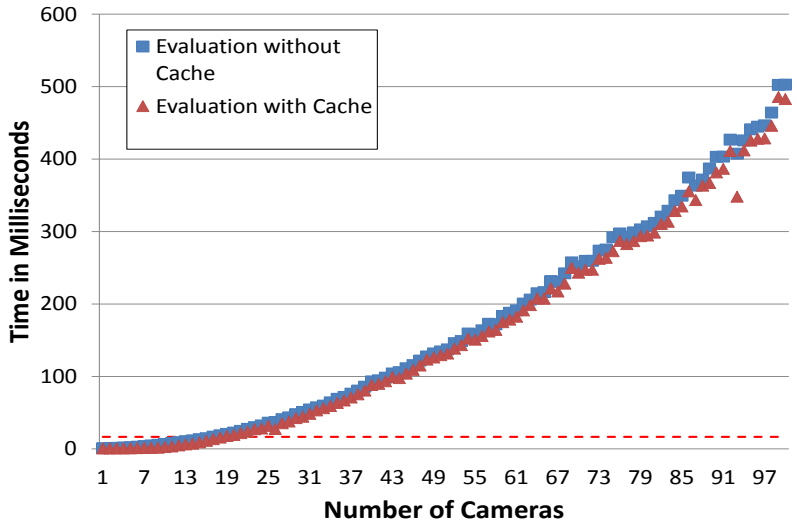

Figure 7.10: Cache vs. No Cache with Two Displays, constraints *KeepCentered*, *DistanceToCamera*, *FrameCoherence*, and *GoalScored*

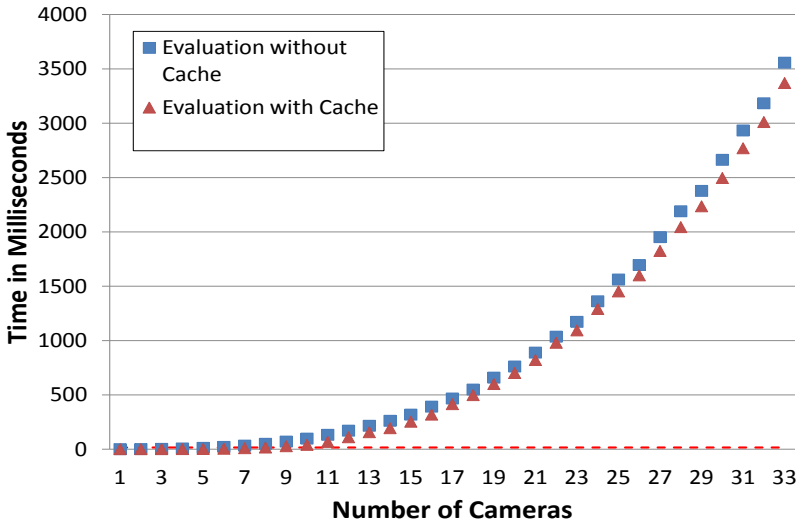

Figure 7.11: Cache vs. No Cache with Three Displays, constraints *KeepCentered*, *DistanceToCamera*, *FrameCoherence*, and *GoalScored*

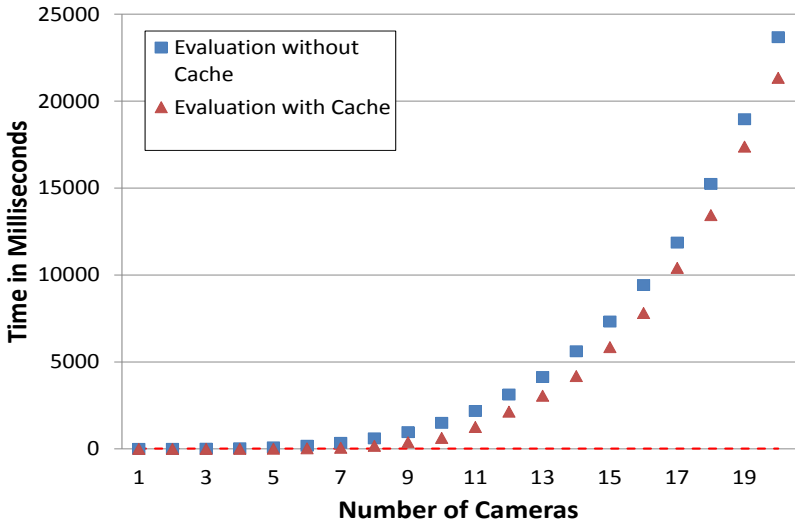

Figure 7.12: Cache vs. No Cache with Four Displays, constraints *KeepCentered*, *DistanceToCamera*, *FrameCoherence*, and *GoalScored*

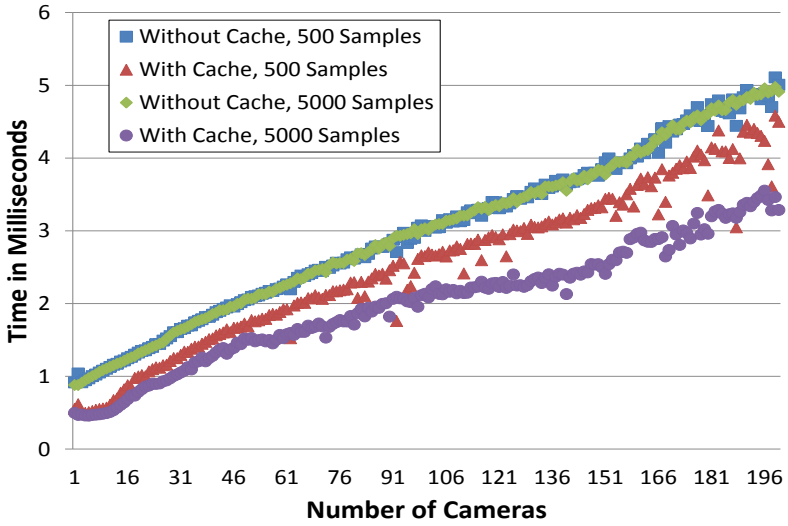

Figure 7.13: Cache vs. No Cache with One Display, 5000 samples per simulation, constraints *KeepCentered*, *DistanceToCamera*, *FrameCoherence*, and *GoalScored*

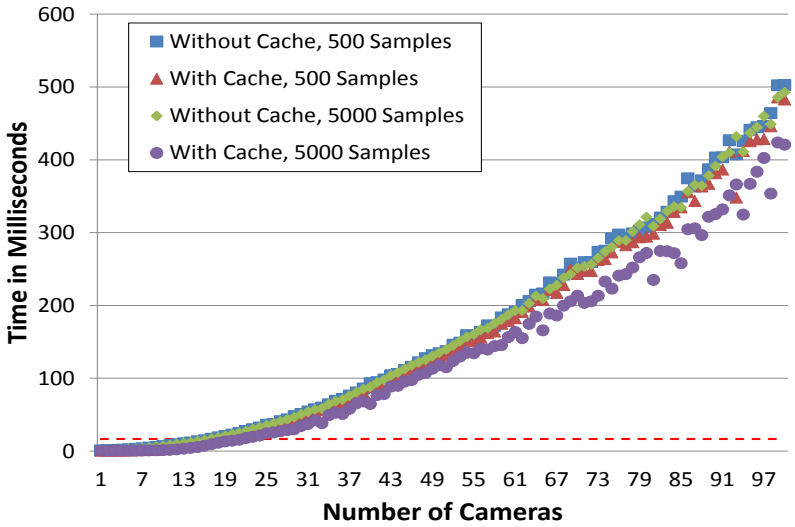

Figure 7.14: Cache vs. No Cache with Two Displays, 5000 samples per simulation, constraints *KeepCentered*, *DistanceToCamera*, *FrameCoherence*, and *GoalScored*

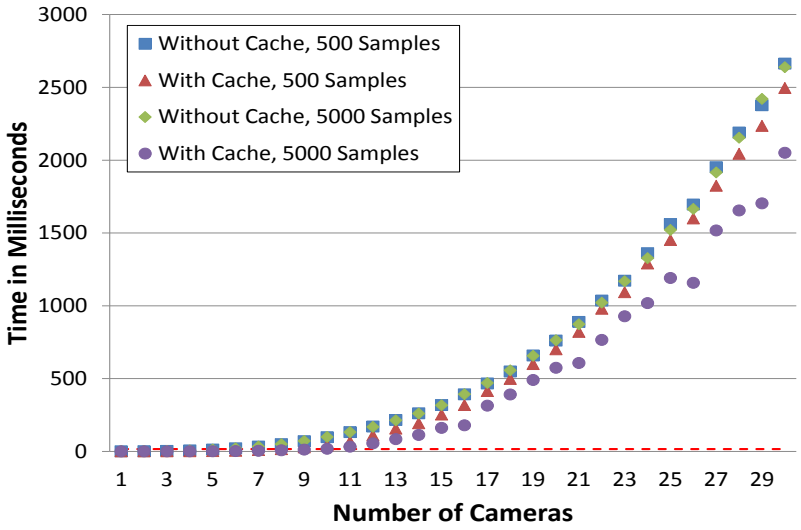

Figure 7.15: Cache vs. No Cache with Three Displays, 5000 samples per simulation, constraints *KeepCentered*, *DistanceToCamera*, *FrameCoherence*, and *GoalScored*

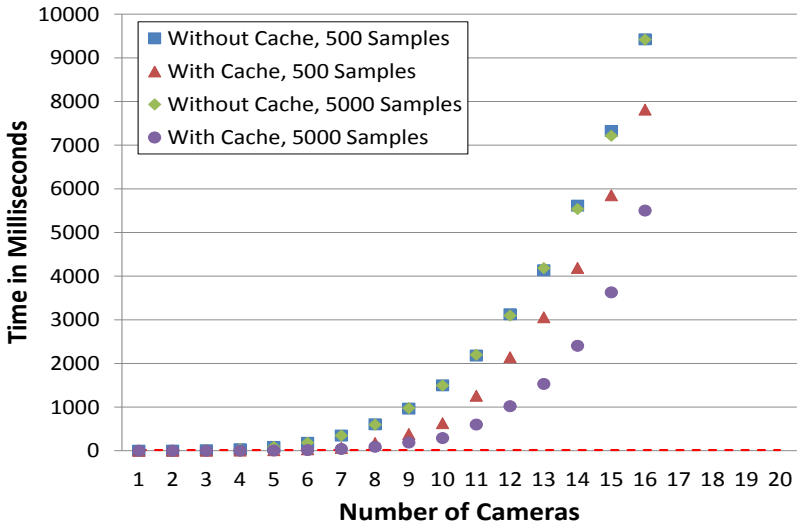

Figure 7.16: Cache vs. No Cache with Four Displays, 5000 samples per simulation, constraints *KeepCentered*, *DistanceToCamera*, *FrameCoherence*, and *GoalScored*

7.2 Native Code Systems

One brute force way to design a camera selection system is to encode the camera selection system in a series of if-then-else statements. These could potentially execute more quickly than searching for the best feed selection. Maintaining large if-then structures is likely to be difficult and error prone, just as maintaining large expert systems are difficult to maintain and extend [52].

An advantage of using a SCSP approach over native code is that the designer can focus on camera selection at an abstract level; for example, specifying that the puck remain in the center of the camera's field of view, instead of using the puck's x, y, and z coordinates directly. Also, a SCSP automatically balances constraints when not all constraints can be simultaneously satisfied, rather than require the designer to write rules for over-constrained situations. Thirdly, preferences are easy to modify by changing them in a small table. With native code, changing one if-statement can have unexpected interactions.

7.3 Conversion to Native Code via Offline Computation

To achieve the potential benefits of both approaches, the SCSP system can be converted automatically to native code. Thus, the camera selection system is designed under the SCSP scheme, but potentially executes in less time.

A naïve conversion would consider all dynamic constraint combinations and map them to the appropriate camera feed. However, the number of states in the input space is prohibitive, even for relatively few constraints. Considering only *KeepCentered* and *DistanceToCamera*, with ten cameras and one display, results in a search space of 3^{20} tuples. If each input is evaluated every 0.006888 milliseconds (from Table 7.1) it would take almost 7 hours to evaluate all input combinations. Adding additional constraints, however, increases the number of combinations, and thus evaluation time, with exponential complexity, assuming the new constraints contain new variables. Using the four constraints and one display as in Figure 7.9, there are $3^{20} \times 6^{10} \times 2^{10}$ tuples in the search space, and would require a approximately 130,000,000 years of computation (at 0.0189822 milliseconds per evaluation, from Table 7.3). Adding more displays would increase the time taken to evaluate each input tuple, but would not increase the search space since more displays do not increase the number of dynamic constraints.

In practice, relatively few dynamic constraint combinations occur. Some combinations are self contradictory and never occur. For example, the puck being in the center of camera one's field of view may mean that the puck is not simultaneously in the center of camera ten's field of view. Likely dynamic constraint combinations are generated by sampling game execution, and stored in a cache large enough to avoid cache conflicts. This avoids a situation where an existing tuple is

removed from the cache. The samples form a training set from which a decision tree is generated using an information gain heuristic. Entropy in a data set is computed as follows [65][5]:

$$I = -\sum_{i=1}^{n} c_i \times \log_2 (c_i)$$

where n is the number of camera feed combinations and c_i is the relative frequency that the given combination occurs. For example, if you consider the possible outcomes of a fair coin, then the entropy is one, since the probability of heads is one half (c_1) and the probability of tails is one half (c_2). When constructing the decision tree, each variable is considered according to how much it reduces the information in the data set given the variable. When considering a variable, the training data set is split depending on the value assigned to the variable. Information is computed for each of the data sets and combined by averaging the information values computed. The variable that results in the minimal amount of information is selected as the next node in the decision tree.

The decision tree is written to C code if-statements which are compiled to native code, which can be done offline. In testing the native code, the size of the dynamic linking library containing the native code remained reasonable, such as 42 kilobytes based on 100 samples and 478 kilobytes based on 5000 samples. When the native code is provided with dynamic constraints that were not in the sampling, it returns the most frequently used camera feed in the decision tree's subtree.

Figures 7.17, 7.18, and 7.19 compare the quality of the native code camera selection to the quality of the SCSP camera selection, with a sequentially increasing number of constraints. Notice that the quality of native code camera selection approaches the quality of the SCSP system asymptotically. Consequently, a high performance can be achieved in a relatively short time period as shown in Figure 7.17 which uses the *KeepCentered* and *DistanceToCamera* preferences with one to five displays. Notice from Figure 7.17 that the number of displays has little effect on the quality of the solution. With large SCSP systems the collection of cache samples can be done in parallel, as can the conversion to native code, to reduce conversion time. The time to retrieve a value from the decision tree is very short as the time is bounded by $O(n)$ where n is the number of variables included in dynamic constraints used in the simulation. In practice this retrieval time is negligible even with a large number of cameras and displays.

As constraints are added, such as in Figures 7.18 and 7.19, the sample space increases in size but the asymptotic nature remains. Figure 7.18 shows the data from one display and two constraints from Figure 7.17 in blue, and three constraints in orange. For testing, multiple decision trees were constructed and converted to native code. The number of samples used to construct the decision tree is shown along the x-axis of the graph. To determine the quality of returned solution, the native code solution was compared to the SCSP solution for 15000 samples. This means that after the decision tree was compiled the resulting native code was used to determine camera feed

[5]Here $log_2(0)$ is assigned 0 to prevent an infinite assignment of information.

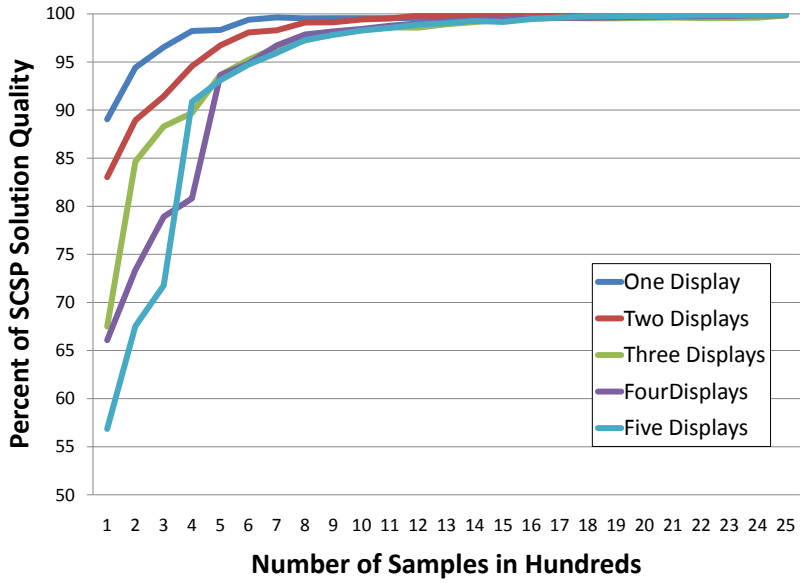

Figure 7.17: Native Code Performance versus optimal SCSP solution (Two Constraints, *KeepCentered*, *DistanceToCamera* plus different camera feed on each display)

selection during a new simulation. Each time the native code made a selection the SCSP solution was run using the same input parameters. The valuation of the SCSP solution was compared to the valuation of the native code solution. For each native code output, the above process was repeated for 15000 camera feed selections and the ratio of the native code valuations to SCSP valuations were averaged to make one data point on the graph.

The search space is very large. For example, the potential number of input combinations is $3^{10} \times 3^{10} \times 6^{10}$ for the constraints *KeepCentered*, *DistanceToCamera*, and *FrameCoherence* when ten cameras are used. Figure 7.19 shows the orange line from Figure 7.18 in blue and the data from testing with four constraints in orange. The potential number of input combinations with the four preferences used is $3^{10} \times 3^{10} \times 6^{10} \times 3^{10} = 12449449430074295092224$ (the *SeeScorer* constraint has been added). Note that 30000 samples is approximately 0.00000000000000024% of the number of input combinations, but on average can select a camera feed with more than 92.5% of the value of the SCSP solution.

As mentioned previously, and as shown from Figures 7.17, 7.18, and 7.19, the number of samples required to achieve a given percentage of the value of the SCSP solution depends primarily on the number of dynamic constraints. While increasing the number of displays increases the number of dynamic constraints, the additional dynamic constraints are correlated (in Figure 7.17 the value of a dynamic constraint for a particular camera is the same on all displays). As the number of displays increases, however, the time required to generate the decision tree also increases. With

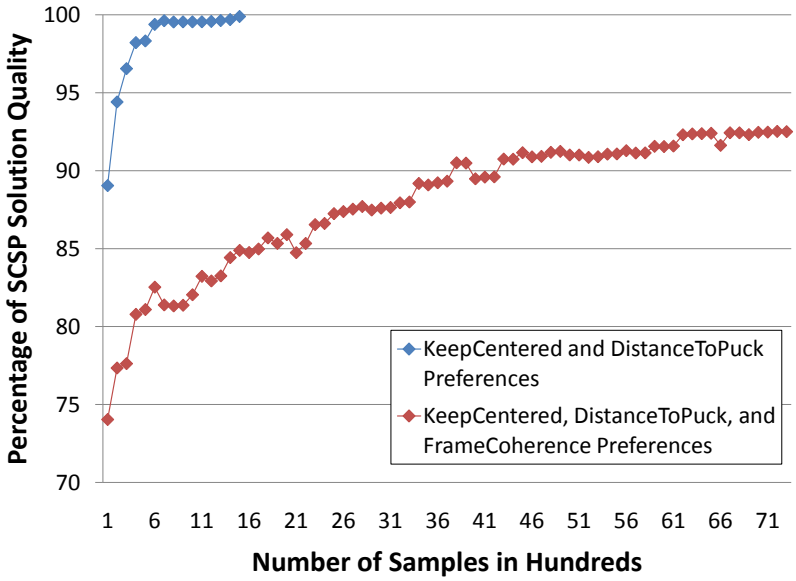

Figure 7.18: Native Code Performance versus SCSP solution (One Display). The blue line ends since the experiment reached approximately 100% of the SCSP solution.

one display and 10 cameras there are 10 camera combinations to select from. However, with two, three, four and five displays there are 100, 1000, 10000, and 100000 camera combinations to select from. Constructing a decision tree from 2500 samples from five displays required approximately half a day of computation using a quad core 3.0GHz Penryn processor[6].

7.4 Chapter Summary

The SCSP camera feed selection system can choose cameras by searching for the tuple with the highest preference, but the search time can become lengthy with a large number of constraints or displays. With one display the number of camera feeds to select among in real time ranges into the thousands, and with two displays the number of camera feeds can reach into the hundreds. When a cache is employed, the average time to select a camera feed decreases, as some tuples can be found in the cache rather than recomputed. The larger the number of tuples in the cache the greater the benefit.

With a large number of cameras, constraints, or displays, the time to determine a camera feed can exceed a real time requirement. In these cases the cache can be converted to native code via a decision tree. The quality of the camera selection from the native code increases asymptotically,

[6]The conversion program was written in Java, which may only use one of the processor's cores.

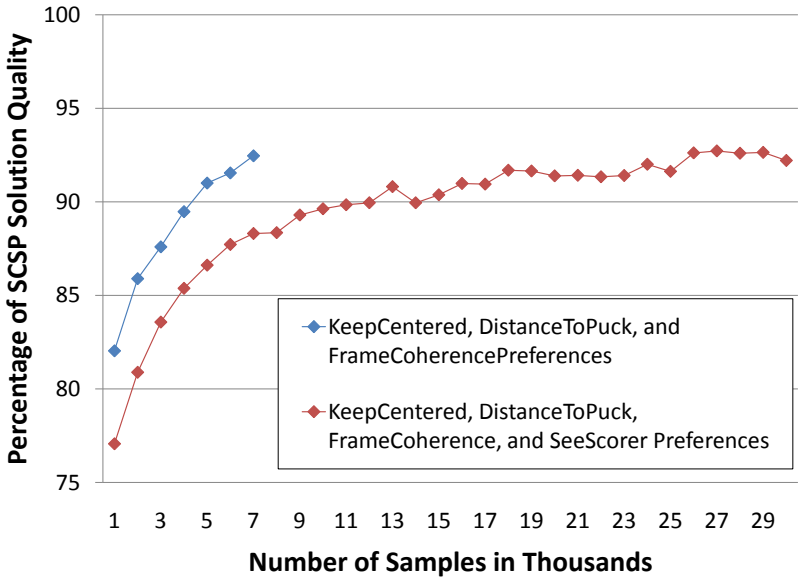

Figure 7.19: Native Code Performance versus SCSP solution (One Display). The blue line in this figure is the orange line in Figure 7.18. The blue line here shows points every thousand samples, instead of hundred samples as in Figure 7.18.

compared to the optimal SCSP solution, as the number of samples used to construct the code increases. Hence a system capable of producing solutions (approximately 92% of the SCSP solution) can be constructed offline, before the final build, from samples composing a small fraction of the overall input space.

Chapter 8

Results

Thus far, this thesis has presented a system to encode a designer's preferences for camera selection and a system to optimize the combination of the individual preference functions. Additionally, if desired, the camera selection system can be converted into a native code solution, resulting in a decrease in camera selection time and an approximate solution.

To demonstrate that, given the input preferences, the system chooses the correct camera feed, this chapter presents resulting screen shots determined by the example preferences. The preferences presented are examples only; a designer may choose different preferences. Section 8.1 uses example preferences with the virtual hockey game, and Section 8.2 uses example preferences with the maze game. Section 8.3 compares a SCSP system with a traditional CSP system to select cameras in the hockey game.

8.1 Examples from the Hockey Game

As a first example preference, a designer may want to keep the puck visually centered in the selected camera's field of view, and to have it appear closer to the camera. It may be that the hockey game is easier to follow if the puck is kept centered in the screen, and it may be easier to see the puck if it is larger. The preferences *KeepCentered* and *DistanceToCamera* in Tables 8.1 and 8.2 provide these directions to the SCSP solver[1].

Location	Preference
center	1.0
border	0.7
out-of-view	0.3

Table 8.1: Example of Static *KeepCentered* Preference

[1] These tables are the same preference functions as Tables 5.5 and 5.9. They are presented here for the reader's convenience.

Distance	Preference
near	1.0
medium	0.8
far	0.5

Table 8.2: Example of Static *DistanceToCamera* Preference

When a solution cannot be found using the highest preference value from each table, the system must determine a trade-off for more fully satisfying one preference function than another. Since multiplication is used for combining preferences, the ratio between preference values (in the same table) determines the decrease in the preference of the solution found. Note the ratios between changing states in the preference tables. Changing from *center* to *border* in *KeepCentered* has an approximate ratio of 1.43, where changing from *near* to *medium* in *DistanceToCamera* has an approximate ratio of 1.25. This means that, in a situation where the SCSP solver must select from two cameras where one camera's state is { *KeepCentered* = *center*, *DistanceToCamera* = *medium* } and the other camera's state is { *KeepCentered* = *border*, *DistanceToCamera* = *near* }, the first camera will be chosen as there is more of a penalty for changing from *center* to *border* as changing from *near* to *medium*.

Figures 8.1 and 8.2 show screen captures of the same game with different preferences. The left image uses only *KeepCentered*, the middle image uses only *DistanceToCamera*, and the right image uses both preferences. Note that the right image in each figure has the puck centered more than the middle image, and the puck closer to the camera than in the left image. The middle image from Figure 8.2 shows a selected camera, where the puck is close to the camera even though the puck is not in the camera's field of view, as this satisfies the *DistanceToPuck*, and the *KeepCentered* is not applied to the middle image. The resulting camera selection, from the preferences *KeepCentered* and *DistanceToCamera*, produces an enjoyable viewing experience for some viewers as it follows the example designer's preferences[2]. As mentioned before, in respect to the ratios between preferences, the system will automatically select a view with the puck centered if a view with the puck centered and close to the camera is not available.

An issue with using only the preferences *KeepCentered* and *DistanceToCamera* is that the camera feed may change too quickly. When the camera feed changes, it seems that a viewer needs a short time to locate the puck on the screen. Additionally, a view may become stale in the sense that changing to another camera feed after a while may add more visual interest. These

[2]What is enjoyable may differ from person to person, but at minimum the author thought the camera selection was beneficial. It allows the action of the game to be followed by keeping the puck in view, and zoomed in when possible.

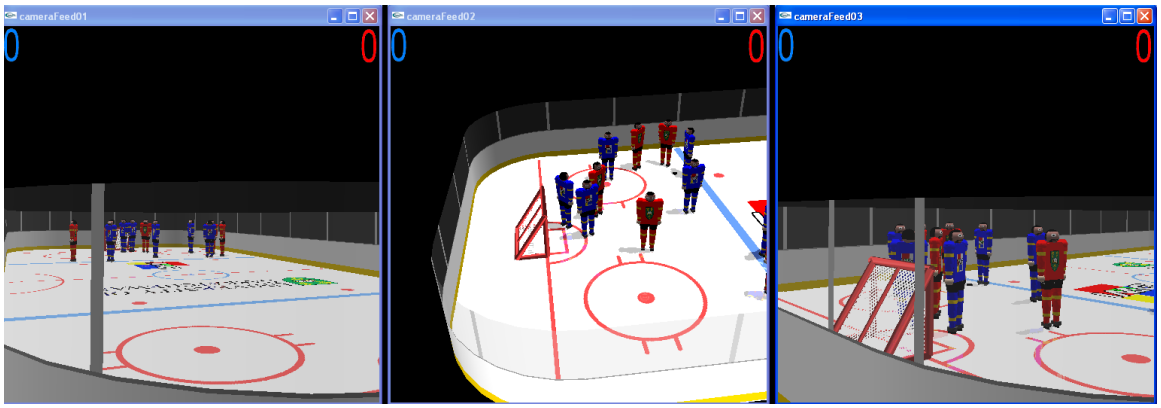

Figure 8.1: Screenshots from using *KeepCentered*, *DistanceToCamera*, and both constraints, resulting in the left, middle and right images, respectively. Notice the puck is centered in the left image, close to the camera in the middle image, and centered and close to the camera in the right image, correctly applying the input preferences.

two considerations, not changing too quickly and not remaining on one camera feed for too long, can be encoded into a preference named *FrameCoherence* (already defined in Section 5.4). Figures 8.3 and 8.4 show the use of the *FrameCoherence* constraint. Figure 8.3 shows nine seconds of play using *KeepCentered* and *DistanceToCamera* constraints, while Figure 8.4 shows the same nine seconds of play, adding *FrameCoherence* as a constraint. It can be seen from the images that the camera feed changes more often in Figure 8.3 than Figure 8.4. Frames labeled five through nine in Figure 8.3 show the camera feed changing every second, while in Figure 8.4 the feed changes between frames labeled one and two, and frames labeled seven and eight. This illustrates the effect of a higher preference for remaining on the same feed when the camera has been selected for five seconds or less, and a lower preference for remaining on the same feed for longer than five seconds. The choice of five seconds is arbitrary and depends on the implementation of the constraint and the designer's preference values. With the example constraints, the system typically implements a trade-off between satisfying the following constraints: selecting a camera feed where the puck is centered, selecting a camera feed where the puck is close to the camera, and not changing the camera feed too frequently. The result is a game that is similar to using only the former two constraints but with the effect of remaining with the same camera feed for a reasonable length of time.

8.2 Examples from the Maze Game

The SCSP solver in the maze game also selects cameras from the tuple with the highest preference. When possible, the avatar is kept in the field of view and the camera adjusts to show a target box or the firing box. When the avatar is placed just in front of a wall the camera changes to show a front

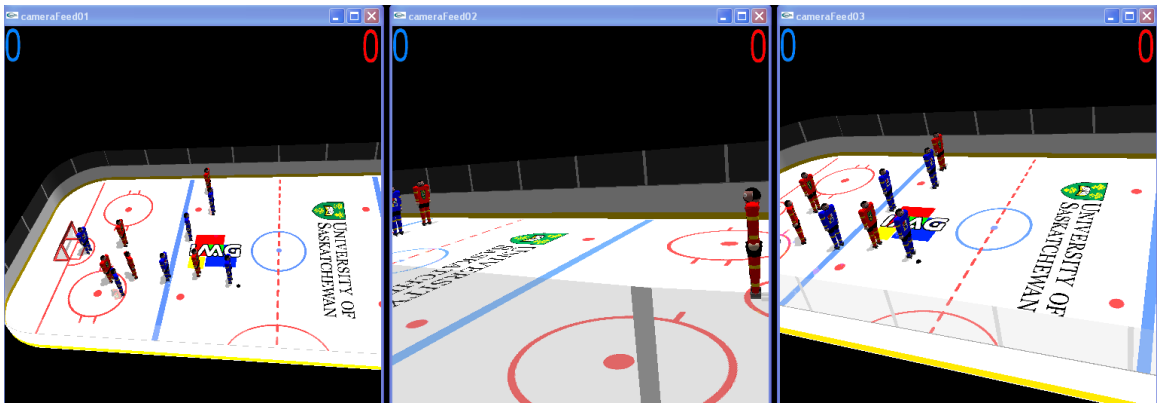

Figure 8.2: Screenshots from using *KeepCentered*, *DistanceToCamera*, and both constraints, resulting in the left, middle, and right images, respectively. In the middle image, even though the puck is out-of-view of the camera's field of view, the puck is close to the camera.

view of the avatar, as opposed to showing a view with the wall blocking the avatar. Transitions through walls are avoided when possible. In an over-constrained case, where the camera cannot simultaneously show a firing box and a target box, the camera follows the firing box, as such an action has a higher associated preference. When the firing box hits a wall, or disappears at its expiry time, the selected camera feed changes to show the target box again.

The maze game is included in this thesis to demonstrate that the camera selection system functions in a first person shooter game (the maze game) in addition to a game with a spectator viewpoint (the hockey game). While the camera selection system correctly implements the preferences, the avatar is difficult to control and aiming at targets is difficult from positions other than behind the avatar. One solution to making the avatar easier to control is to implement the preference to always select the camera feed that is behind the avatar. While this is a valid constraint, it does not demonstrate the benefit of using the camera selection system, as the system will always choose the same camera. Another solution, to provide the benefits of easier control of the avatar and showing target boxes, is to use a two display system as shown in Figure 8.5. The larger view provides a viewpoint from behind the avatar. The inset view in Figure 8.5 selects a camera feed that provides hints to the user, in this case that a target box is to the avatar's right. If a user has two screens, the inset view can be shown on the second monitor.

When playing the maze game with two screens the avatar can be controlled from the main screen (with a viewpoint from behind the avatar) and the user can look at the hint display to determine the next goal of the game. For example, sometimes after a target box is hit with a firing box, the hint camera changes to a viewpoint showing the next target box. In another example, when the avatar is backed against a wall, the main display shows the wall (since the wall occludes

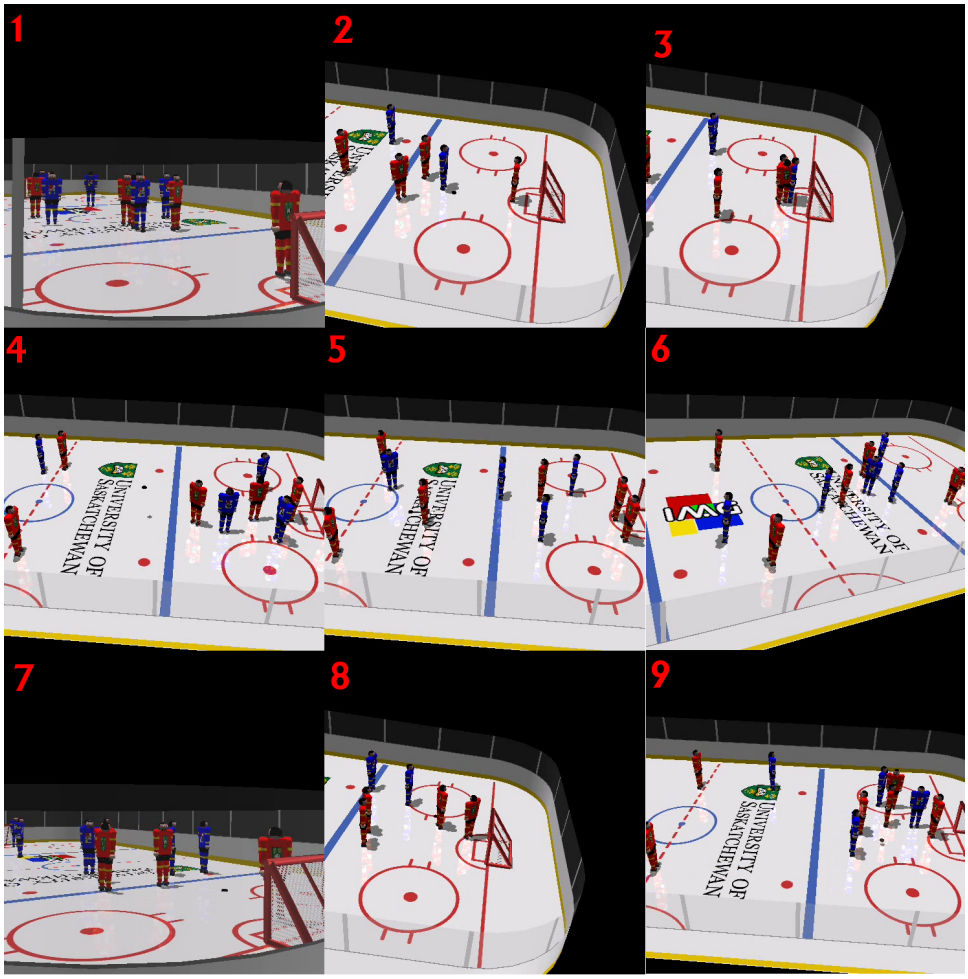

Figure 8.3: Screenshots from nine seconds of play without frame coherence. Notice the frequent camera changes between frames.

the avatar), while the hint camera changes to show the front of the avatar. By looking at the hint display, the user can determine when to move forward to move away from the wall. Thus, using different preferences for each display can combine the effects for easier control of the avatar, and show key game elements (such as targets).

8.3 Evaluation: CSP versus SCSP solutions

Camera placement methods have used CSPs with hard constraints, and He *et. al*'s Virtual Cinematographer [49] uses hard constraints, in that guards on transitions between states are either satisfied or unsatisfied. Since there are no implementations of these systems available, this chapter presents a comparison between a hard constraint system (CSP) and the soft constraint system introduced in this work (SCSP).

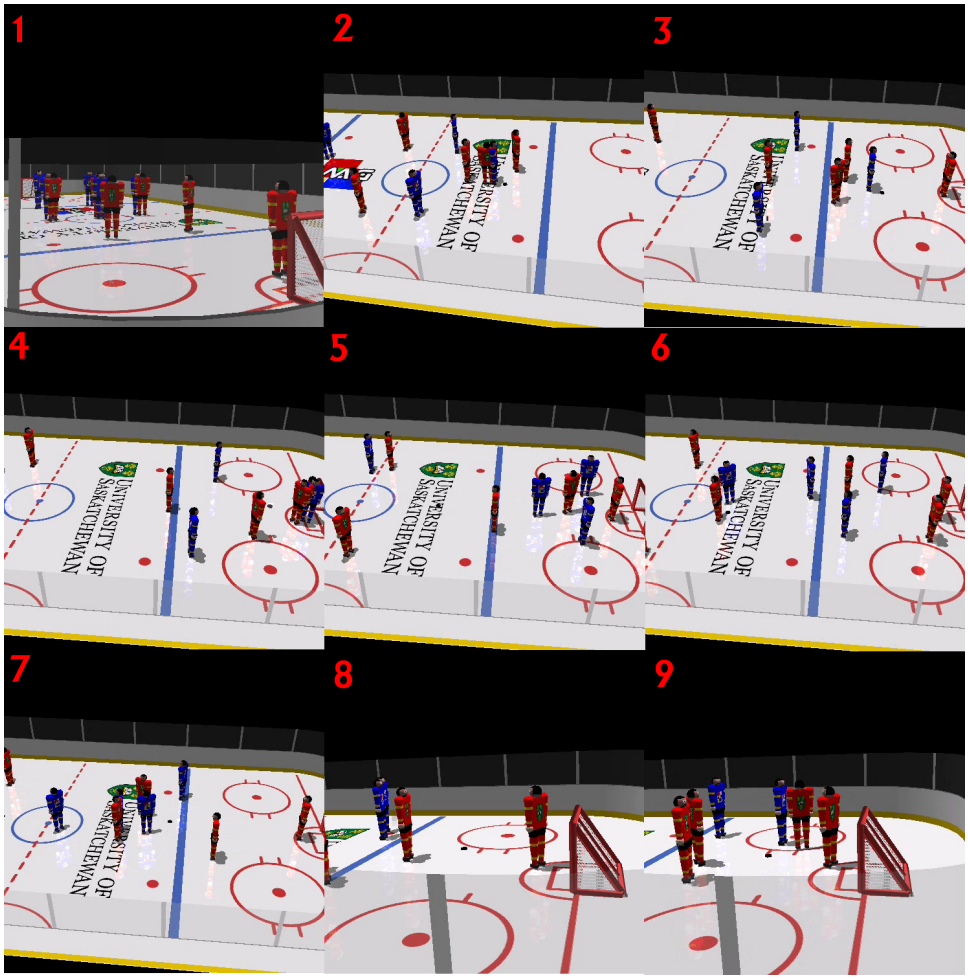

Figure 8.4: Screenshots from nine seconds of play with frame coherence. Notice the same camera is selected from frames two through seven. After five seconds there is less of a penalty for changing to another camera, explaining the camera change between frames seven and eight.

The SCSP camera selection system can be viewed as a tool. The question of whether or not this tool is better than a previous tool is similar to asking whether a new paint brush can paint a better picture than a previous paint brush. In the case of a new paint brush, whether or not a painted picture is better depends on the skill of the artist. Similarly, whether the SCSP camera selection system performs better that a previous system depends on the designer's goals. Additionally, "better" in this argument is a subjective judgement and depends on the person doing the evaluating. Consequently, rather than demonstrate that a new tool is better, this section will show that the SCSP camera selection system is useful, mainly that the SCSP system can handle over-constrained problems, where systems based on hard constraints cannot handle over-constrained problems. Note that the SCSP camera selection system was not tested directly against previous implementations

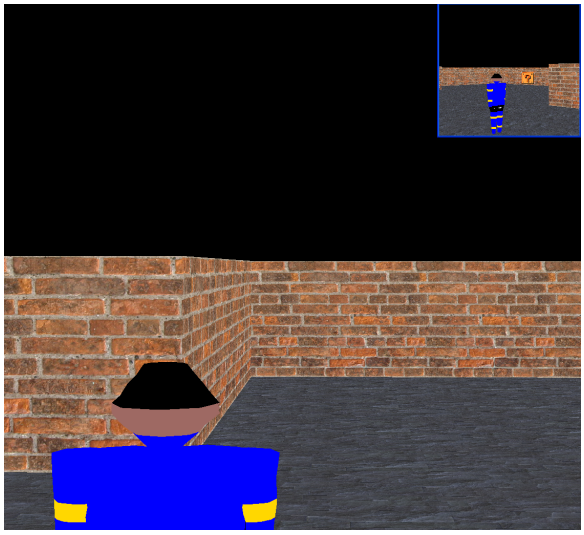

Figure 8.5: Maze Screen shot showing camera behind avatar, and hint camera

such as the Virtual Cinematographer but, for comparison purposes, preferences are mapped to hard constraints as the example shown in Table 8.3.

Location	*SCSP Pref*	CSP Value
center	1.0	*true*
border	0.7	*true*
out	0.3	*false*

Table 8.3: Example SCSP to CSP mapping for testing

Similarly, for Tables 5.9 and 5.10 values 0.5 and greater are mapped to *true* and values less than 0.5 are mapped to *false*. Other mappings are possible. For example, the designer may or may not consider a preference of 0.7 to satisfy the constraint.

For each number of displays, the hockey game was run with *KeepCentered* and *DistanceTo-Camera* for at least 10000 camera feed selections. Additional static constraints preferred different camera feeds on each screen. The tests were repeated with an added *FrameCoherence* constraint. The number of times the hard constraint system found a solution was recorded and compared to the total number of searches. The results are shown as a percentage in Table 8.4. As Table 8.4 shows, the CSP method frequently finds solutions when the problem is under-constrained (such as with one display) but finds solutions less often as the problem becomes over-constrained. When the system is executed using three constraints and five displays, the CSP method finds no solutions.

The SCSP solution was able to find a solution 100% of the time, albeit by partially satisfying

Display Count	Two Constraints	Three Constraints
1	98.4%	91.5%
2	91.6%	69.2%
3	65.4%	35.6%
4	44.0%	18.4%
5	3.4%	0.0%

Table 8.4: Percent of Samples CSP solved

constraints as the system became over-constrained. Notice in Table 8.4 that the percentage of solutions decreases with the same number of displays when the number of constraints increases from two to three, a trend that continues as additional constraints are added to the system.

8.4 Chapter Summary

The hockey and maze games are able to use a designer's preferences as input, and produce the corresponding camera selection. In the hockey game, preference values for the constraints *Keep-Centered* and *DistanceToCamera* keep the puck in the center of the camera's field of view and select views where the puck is larger in the camera's field of view. An additional *FrameCoherence* constraint can prevent the system from changing camera feeds too frequently or from remaining on one camera feed for too long. When the problem is over-constrained, the camera selection system will select the best available solution, determined by trade-offs according to the ratio between preference values. In the maze game, a high preference to select the camera behind the avatar can facilitate aiming the avatar, while a help display can show important game details, such as the next target box. Additionally, when the view of the avatar on the main display is occluded, the hint display can show an unobstructed view of the avatar. Testing shows that the SCSP camera selection system can determine a camera feed in over-constrained problems, where a traditional CSP selection system finds fewer solutions as the problem becomes more constrained.

Chapter 9

Conclusion

Prior to this thesis work, the problem of selecting a virtual camera in a dynamic scene in real time was unsolved. The SCSP camera selection system provides a method to select from multiple cameras in real time, in a dynamic scene, such as a video game. The number of cameras that can be handled in real time depends on the acceptable length for a search. Graphs, in Chapter 7, show the trade-offs between number of cameras, number of displays, and number of constraints. For one display, the number of cameras permitted ranges from hundreds to thousands depending on the constraints used in the search.

The system selects a camera based on static and dynamic constraints. Specifying a designer's static preferences in modular tables makes the preference specification tractable, since the potential search space is exponential in size (Section 6.4). Dynamic constraints, set by the system at run time, cause the SCSP selection system to choose camera configurations that currently exist (as opposed to selecting a prefered configuration that is not available). The SCSP camera selection system is flexible, in that it can handle arbitrary constraints as long as the designer provides the preferences for the constraint, and the corresponding dynamic constraint is implemented in the simulator. Some preferences, such as specifying that two displays use different feeds, do not need a corresponding dynamic constraint. The use of a SCSP camera selection system may allow users to customize the camera selection system after a game's release if the preferences of the camera selection system are exposed to the user.

Modifying and expanding a system based on the SCSP framework can be more manageable than a system based on rules (Section 7.3). When a camera selection system based on rules is desired, perhaps for a faster selection system for a commercial product, the SCSP preferences can be converted to native code. This involves storing searches in a cache. The cache itself can decrease the average search time, since when dynamic constraints are the same as a previous case the result can be retrieved from cache. Alternatively, the cache can be converted to a decision tree, which can be converted to native machine code. The preference value of the native code solution asymptotically approaches the preference value of the SCSP system with the number of samples used to generate the native code. When converting to native code, the specification of preferences is still entered in modular tables under the SCSP framework.

Differing from previous systems, the SCSP camera selection system is robust, in that it can handle over-constrained and under-constrained problems automatically, based on the level of preference provided by the designer. A CSP system is unable to determine a camera feed when the problem is over-constrained, or select from among multiple solutions when the problem is under-constrained unless an objective function over the CSP solutions is used, similar to PCSP systems. However, previous work has used hard CSP systems without such an objective function. In the case of the Virtual Cinematographer by He *et al.*, the system may continue to select a suboptimal camera feed since the guard on a transition is not fully satisfied. In the case of a CSP, the system may simply determine that there is no solution, since not all constraints can be satisfied. The SCSP solution chooses the tuple with the highest available preference, regardless of whether the problem is over-constrained or under-constrained.

This thesis details two example games demonstrating the use of the SCSP camera selection system. The first is a spectator viewpoint hockey game. Constraints implemented in the hockey game include the preference to keep the hockey puck centered in the display, as well as to select cameras where the puck is closer to the camera. Other preferences include showing the scorer after a goal, and frame coherence. Frame coherence is a temporal constraint that decreases the frequency of camera changes, except when the display has remained on the same camera feed for a long time. The second example game is a third person perspective maze game. It uses constraints to prevent the user's avatar from being occluded, as well as keeping the firing and target boxes from being occluded. Since transitions between cameras are linearly interpreted, other constraints prevent the camera from passing through walls.

As shown in Chapter 8, the system can naturally extend to multiple displays. For the hockey simulation, constraints can specify that each screen show a different feed. In the maze game, the first display can allow the user to control their avatar, while a second display functions as a hint camera. The first display offers easier control by showing a viewpoint from behind the avatar. The hint camera can change camera feed to show important events or objects. For example, the hint camera can show a target box that is off screen of the first display. As well, if the first camera's view is occluded by a wall, the hint camera can show the user an unoccluded view of the avatar.

As its main contribution, this work solves the problem of camera selection in dynamic scenes. This system can function in real time, with up to thousands of camera feeds. The number of displays, since they are output variables, most strongly affects the limit on the number of manageable camera feeds, while the number of constraints also affects the number of cameras permitted. For a reduced decision time, if an approximate solution is permitted, the system can use cache or be converted to native code. Furthermore, the SCSP system finds the camera feed corresponding to the highest preference, where previous work may either not return a solution or return a suboptimal solution.

In summary, the SCSP approach provides a tool to select a camera feed in dynamic scenes.

Instead of having a designer explicitly create hand coded rules for the camera to select, as in a rule-based system, the system will combine the designer's preferences automatically. For large systems with many displays and possible camera feeds the time required for the system to return a solution may exceed real time limits. In these cases the SCSP preferences can be automatically converted to a native code system at the cost that the native code system approaches the optimal solution found by the SCSP asymptotically, as the number of samples used to create the native code system approaches the size of the search space.

9.1 Future Work

For future work, the SCSP solution can be extended to new domains. Hockey tends to have the puck as one focus of interest. Other games, such as capture the flag, have simultaneous multiple points of interest. Using multiple displays, each area of interest could be shown to the viewer. Baseball is another game with multiple visual areas of interest. While the pitcher is throwing a ball an opposing player may be trying to steal a base, or the system could show players tagging up when a ball is caught in the outfield. Car racing is another game environment that can have multiple visual points of interest that may be well suited to a SCSP camera selection system.

The SCSP camera selection system may also be expanded to handle selecting feeds from real cameras. In amateur sports, the camera selected may come from a number of fixed cameras or spectators who film the game. Typically, in order for the system to select a camera feed in the virtual hockey game, the puck's location must be known. If this information can be extracted from the environment, then the SCSP camera selection system should function using real footage in a similar fashion to the virtual hockey environment. Other games, such as basketball, should be similar to hockey in that there is usually a single major point of interest.

Additional output variables, other than the camera feed for a particular display, may be added to the system. For example, the system may be able to select the number of displays that are needed at run time. In the maze game the hint camera could appear only when needed and only in a location where it does not obscure important game elements. Also the system could generate replay summaries including the number of cameras needed to display information. When the system generates replays for later viewing, selecting zero displays would indicate that nothing interesting is currently happening and the system should skip to the next scene to show.

Differing from the sports domain, the SCSP camera selection system may be adaptable to produce a rough, or perhaps final, cut of video footage, either real or virtual. For example, multiple YouTube videos of the same event may be combined to show a viewpoint that reflects a designer's aesthetics. If the input is photos instead of video, the system may be able to automatically produce a slide show.

Currently dynamic constraints reflect the state of the simulation with certainty. The system may be expanded to probabilistic dynamic constraints that predict future dynamic constraints with a degree of certainty. For example, the future location of the puck in a display's field of view may have a probability distribution that would then map to a probability distribution for the dynamic constraint.

Also as future work is a further examination of the SCSP camera selection system's preferences in comparison to utility theory. Preferences may be mapped to, or inputted as, utility values. With appropriate SCSP combination operators the system could manipulate utilities, rather than preferences. It is currently unclear how this change would effect the camera feed selected.

Future work could balance competing output interests using this system with only minor modification, such as adding the additional output variables. Examples of such variables include selecting appropriate advertising in the background of a sports game, or removing occluding objects. With respect to advertising, a nearer, or more centered advertisement space may be more valuable. With respect to occlusion, the SCSP approach may determine that it is preferred to remove an occluding player or object (such as a goalie when the play is at the other end of the rink) or to make the player semi-transparent to still present a view of the puck.

References

[1] Md. Shafiul Alam and Scott D. Goodwin. Control of constraint weights for a 2d autonomous camera. In Yong Gao and Nathalie Japkowicz, editors, *Advances in Artificial Intelligence*, volume 5549 of *Lecture Notes in Computer Science*, pages 121–132. Springer Berlin / Heidelberg, 2009.

[2] Edward Angel. *Interactive Computer Graphics: A Top-Down Approach With OPENGL primer package-2nd Edition*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2001.

[3] Daniel Arijon. *Grammar of the Film Language*. Silman-James Press, 1976.

[4] J. Assa, L. Wolf, and D. Cohen-Or. The Virtual Director: a Correlation-Based Online Viewing of Human Motion. *Computer Graphics Forum*, 29:595–604, 2010.

[5] W. H. Bares and J. C. Lester. Intelligent Multi-Shot Visualization Interfaces for Dynamic 3D Worlds. In *Intelligent User Interfaces*, pages 119–126, 1999.

[6] William Bares and Byungwoo Kim. Generating virtual camera compositions. In *IUI '01: Proceedings of the 6th International Conference on Intelligent User Interfaces*, pages 9–12, New York, NY, USA, 2001. ACM Press.

[7] William Bares, Scott McDermott, Christina Boudreaux, and Somying Thainimit. Virtual 3D Camera Composition from Crame Constraints. In *MULTIMEDIA '00: Proceedings of the Eighth ACM International Conference on Multimedia*, pages 177–186, New York, NY, USA, 2000. ACM Press.

[8] Prabir Bhattacharya, Azriel Rosenfeld, and Isaac Weiss. *Point-to-line mappings and Hough transforms*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[9] S. Bistarelli, U. Montanari, F. Rossi, T. Schiex, G. Verfaillie, and H. Fargier. Semiring-Based CSPs and Valued CSPs : Frameworks, Properties and Comparison. *Constraints*, 4:199–240, 1999.

[10] Stefano Bistarelli, Eugene C. Freuder, and Barry O'Sullivan. Encoding Partial Constraint Satisfaction in the Semiring-Based Framework for Soft Constraints. In *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence*, ICTAI '04, pages 240–245, Washington, DC, USA, 2004. IEEE Computer Society.

[11] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Constraint solving over semirings. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1*, pages 624–630, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[12] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44:201–236, 1997.

[13] Jim Blinn. Where am I? What am I looking at? *IEEE-CGA*, 8(4):76–81, July 1988.

[14] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint Hierarchies. *LISP and Symbolic Computation*, 5:223–270, 1992. 10.1007/BF01807506.

[15] Alan Borning, Amy Martindale, Molly Wilson, and Michael Maher. Constraint hierarchies and logic programming. In *In Proceedings of the Sixth International Conference on Logic Programming*, pages 149–164. MIT Press, 1989.

[16] Owen Bourne, Abdul Sattar, and Scott Goodwin. A Constraint-Based Autonomous 3D Camera System. *Constraints*, 13(1-2):180–205, 2008.

[17] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio link frequency assignment. *Constraints*, 4:79–89, 1999. 10.1023/A:1009812409930.

[18] David B. Christianson, Sean E. Anderson, Li-wei He, David Salesin, Daniel S. Weld, and Michael F. Cohen. Declarative camera control for automatic cinematography. In *AAAI/IAAI, Vol. 1*, pages 148–155, 1996.

[19] M. Christie and H. Hosobe. Through the Lens Cinematography. In *Proceedings of the 6th International Symposium on Smart Graphics*, pages 147–159, 2006.

[20] M. Christie, Rumesh Machap, Jean-Marie Normand, Patrick Olivier, and Jonathan Pickering. Virtual Camera Planning: A Survey. In *Proceedings of the 5th International Symposium on Smart Graphics*, pages 40–52, 2005.

[21] Marc Christie and Patrick Olivier. Camera control in computer graphics: models, techniques and applications. In *ACM SIGGRAPH ASIA 2009 Courses*, SIGGRAPH ASIA '09, pages 3:1–3:197, New York, NY, USA, 2009. ACM.

[22] M. Cooper, S. De Givry, M. Sanchez, T. Schiex, and M. Zytnicki. Virtual Arc Consistency for Weighted CSP. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 1*, pages 253–258. AAAI Press, 2008.

[23] Martin Cooper and Thomas Schiex. Arc consistency for soft constraints. *Artificial Intelligence*, 154(1-2):199 – 227, 2004.

[24] Martin C. Cooper. Reduction operations in fuzzy or valued constraint satisfaction. *Fuzzy Sets Syst.*, 134:311–342, March 2003.

[25] Martin C. Cooper. High-order consistency in valued constraint satisfaction. *Constraints*, 10:283–305, July 2005.

[26] M.C. Cooper, S. de Givry, M. Sanchez, T. Schiex, M. Zytnicki, and T. Werner. Soft arc consistency revisited. *Artificial Intelligence*, 174(7-8):449 – 478, 2010.

[27] Simon de Givry, Federico Heras, Matthias Zytnicki, and Javier Larrosa. Existential arc consistency: Getting closer to full arc consistency in weighted CSPs. In *IJCAI'05*, pages 84–89, 2005.

[28] R. Dechter. On the expressiveness of networks with hidden variables. In *Proc. of AAAI-90*, pages 556–562, Boston, MA, 1990.

[29] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artif. Intell.*, 34(1):1–38, 1987.

[30] Rina Dechter. *Constraint Processing.* Morgan Kaufmann Publishers, 2003.

[31] Rina Dechter and Peter van Beek. Local and global relational consistency. *Theor. Comput. Sci.*, 173(1):283–308, 1997.

[32] Steven M. Drucker. *Intelligent Camera Control for Graphical Environments.* PhD thesis, MIT, MIT Media Lab, 1994.

[33] Steven M. Drucker, Li-wei He, Michael Cohen, Curtis Wong, and Anoop Gupta. Spectator games: A new entertainment modality for networked multiplayer games. Technical report, Microsoft Research, 2000.

[34] Steven M. Drucker and David Zeltzer. Intelligent Camera Control in a Virtual Environment. In *Proceedings of Graphics Interface '94*, pages 190–199, Banff, Alberta, Canada, 1994.

[35] Steven M. Drucker and David Zeltzer. Camdroid: A system for implementing intelligent camera control. In *Symposium on Interactive 3D Graphics*, pages 139–144, 1995.

[36] D. Dubois, H. Fargier, and H. Prade. The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In *Proc. 2nd IEEE Conference on Fuzzy Systems*, pages 1131–1136, San Francisco, CA, March 1993.

[37] Eidos Interactive. Tomb raider, developed by Core Design, Derby, England, 1996.

[38] Electronic Arts Inc. The Sims 2, Maxis, Redwood City, CA, 2006.

[39] Steven Feiner and Dorée D. Seligmann. Cutaways and ghosting: satisfying visibility constraints in dynamic 3D illustrations. *The Visual Computer*, 8(5&6):292–302, 1992.

[40] Eugene C. Freuder. A sufficient condition for backtrack-free search. *J. ACM*, 29(1):24–32, 1982.

[41] Eugene C. Freuder and Richard J. Wallace. Partial Constraint Satisfaction. *Artif. Intell.*, 58(1-3):21–70, 1992.

[42] John Gary Gaschnig. *Performance measurement and analysis of certain search algorithms.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1979.

[43] Aditya Ghose and Peter Harvey. Metric SCSPs: Partial Constraint Satisfaction via Semiring CSPs augmented with metrics. In *Proceedings of the 15th Australian Joint Conference on Artificial Intelligence: Advances in Artificial Intelligence*, AI '02, pages 443–454, London, UK, UK, 2002. Springer-Verlag.

[44] M. Gleicher and A. Witkin. Through-the-Lens Camera Control. *Computer Graphics*, 26(2):331–340, 1992.

[45] H. Gouraud. Continuous shading of curved surfaces. *IEEE Trans. Comput.*, 20(6):623–629, 1971.

[46] Mark Haigh-Hutchinson. *Real-Time Cameras: A Guide for Game Designers and Developers.* Morgan Kaufmann Publishers, 2009.

[47] Nicolas Halper, Ralf Helbing, and Thomas Strothotte. A Camera Engine for Computer Games: Managing the Trade-Off Between Constraint Satisfaction and Frame Coherence. In A. Chalmers and T.-M. Rhyne, editors, *EG 2001*, volume 20(3), pages 174–183. Blackwell Publishing, 2001.

[48] R. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980.

[49] Li-wei He, Michael F. Cohen, and David H. Salesin. The Virtual Cinematographer: A Paradigm for Automatic Real-Time Camera Control and Directing. *Computer Graphics*, 30(Annual Conference Series):217–224, 1996.

[50] Donald D. Hearn and M. Pauline Baker. *Computer Graphics with OpenGL.* Prentice Hall Professional Technical Reference, 2003.

[51] Michael C. Horsch, William S. Havens, and Aditya Ghose. Generalized Arc Consistency with Application to MaxCSP. In *AI '02: Proceedings of the 15th Conference of the Canadian Society for Computational Studies of Intelligence on Advances in Artificial Intelligence*, pages 104–118, London, UK, 2002. Springer-Verlag.

[52] Peter Jackson. *Introduction to Expert Systems.* Addison-Wesley, 1986.

[53] Francis S. Hill Jr. and Stephen M Kelley. *Computer Graphics Using OpenGL (3rd Edition).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[54] Kevin Kennedy and Robert E. Mercer. Planning animation cinematography and shot structure to communicate theme and mood. In *SMARTGRAPH '02: Proceedings of the 2nd International Symposium on Smart Graphics*, pages 1–8, New York, NY, USA, 2002. ACM Press.

[55] Vipin Kumar. Algorithms for Constraint Satisfaction problems: A Survey. *AI Magazine*, 13(1):32–44, 1992.

[56] A. H. Land and A. G Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[57] Javier Larrosa. Node and Arc Consistency in Weighted CSP. In *Eighteenth National Conference on Artificial Intelligence*, pages 48–53, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[58] Javier Larrosa and Pedro Meseguer. Optimization-based Heuristics for Maximal Constraint Satisfaction. In *CP*, pages 103–120, 1995.

[59] Javier Larrosa and Thomas Schiex. In the quest of the best form of local consistency for Weighted CSP. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 239–244, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

[60] Louise Leenen. *Solving semiring constraint satisfaction problems.* PhD thesis, University of Wollongong, Faculty of Informatics, 2010.

[61] Louise Leenen, Thomas Meyer, and Aditya Ghose. Relaxations of semiring constraint satisfaction problems. *Inf. Process. Lett.*, 103:177–182, August 2007.

[62] C. Lino, M. Christie, F. Lamarche, G. Schofield, and P. Olivier. A Real-time Cinematography System for Interactive 3D Environments. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '10, pages 139–148, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[63] Alan K. Mackworth. Consistency in networks of relations. *Artif. Intell.*, 8(1):99–118, 1977.

[64] Joseph V. Mascelli. *The Five C's of Cinematography: Motion Picture Filming Techniques.* Cine/Grafic Publications, 1965.

[65] Tom M. Mitchell. *Machine Learning.* McGraw-Hill, 1997.

[66] Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Inf. Sci.*, 7:95–132, 1974.

[67] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[68] Rick Parent. *Computer animation: algorithms and techniques.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[69] Erick B. Passos, Anselmo Montenegro, Esteban W. G. Clua, Cezar Pozzer, and Vinicius Azevedo. Neuronal editor agent for scene cutting in game cinematography. *Comput. Entertain.*, 7(4):1–17, 2009.

[70] J. Pearl. Fusion, Propagation, and Structuring in Belief Networks. *Artif. Intell.*, 29(3):241–288, 1986.

[71] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, 1975.

[72] P. Prosser. Hybrid algorithms for constraint satisfaction problems. *Computational Intelligence*, 9(3):268–299, 1993.

[73] Jean-Charles Régin. AC-*: A configurable, generic and adaptive arc consistency algorithm. In Peter van Beek, editor, *CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 505–519. Springer, 2005.

[74] Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus one-way constraints in user interfaces: experience with the deltablue algorithm. *Softw. Pract. Exper.*, 23:529–566, May 1993.

[75] Thomas Schiex. Arc Consistency for Soft Constraints. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming*, CP '02, pages 411–424, London, UK, 2000. Springer-Verlag.

[76] Thomas Schiex, Helene Fargier, and Gerard Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In *Proc. of the International Joint Conference in AI, Montreal, Canada*, pages 631–637, August 1995.

[77] M. Segal and K. Akeley. The Design of the OpenGL Graphics Interface, 1994.

[78] Dorée Duncan Seligmann and Steven Feiner. Automated generation of intent-based 3D Illustrations. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 123–132, New York, NY, USA, 1991. ACM Press.

[79] L. Shapiro and R. Haralick. Structural Descriptions and Inexact Matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3:504–519, 1981.

[80] Ken Shoemake. Animating rotation with quaternion curves. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 245–254, New York, NY, USA, 1985. ACM Press.

[81] Bill Tomlinson, Bruce Blumberg, and Delphine Nain. Expressive autonomous cinematography for interactive virtual environments. In *AGENTS '00: Proceedings of the Fourth International Conference on Autonomous Agents*, pages 317–324, New York, NY, USA, 2000. ACM Press.

[82] C. Turkay, E. Koc, and S. Balcisoy. An information theoretic approach to camera control for crowded scenes. *Vis. Comput.*, 25(5-7):451–459, 2009.

[83] Pere-Pau Vázquez. Automatic view selection through depth-based view stability analysis. *Vis. Comput.*, 25(5-7):441–449, 2009.

[84] Richard J. Wallace and Eugene C. Freuder. Conjunctive Width Heuristics for Maximal Constraint Satisfaction. In *AAAI*, pages 762–768, 1993.

[85] Turner Whitted. An improved illumination model for shaded display. *SIGGRAPH Comput. Graph.*, 23(6):7, 1980.

[86] Molly Wilson and Alan Borning. Hierarchical constraint logic programming. In *Journal of Logic Programming*, pages 227–318, 1993.

Appendix A

A Program for Designing SCSP Constraints

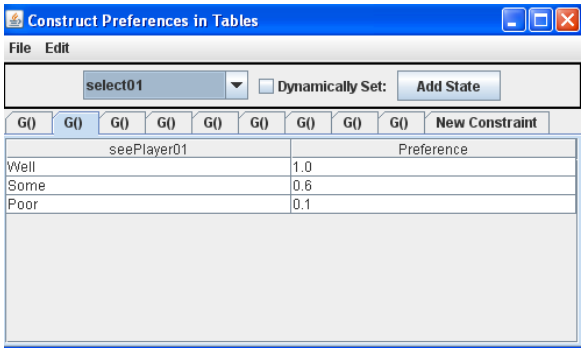

Figure A.1: Java Program for Constructing Constraints

In order to make entering static and dynamic constraints easier for a designer, a java program, shown in Figure A.1, provides a GUI to write the preference XML file. As input the java program takes a C++ file where variables for the constraints are marked with a *visibleVariable* type, which is a pointer to an array of double values (#define visibleVariable double *). A comment line below the line starting with *visibleVariable* lists the states in the domain of the variable. If the function marked with a *visibleVariable* return type is for a dynamic constraint, then it is numbered so that it can be called by number using a function pointer array. Thus, the java GUI allows the designer/director to construct static preferences with preference values, and dynamic preferences that the system will call by number at run time.

The code below shows the camera feed for display one with 10 states, which is used in static constraints.

```
visibleVariable cameraFeed01()
/*Cam0,Cam1,Cam2,Cam3,Cam4,Cam5,Cam6,Cam7,Cam8,Cam9*/
{
  return NULL;
}
```

Usually the GUI will create a variable for each potential display to break potential name dependencies. For example, *KeepCentered* will have *location01* .. *location_n* variables where n is the number of potential displays. However, if a comment listing *only one* is provided then there will only be one variable available in the GUI as shown for the *goalScored* variable/function. *goalScored* may be used either in a static constraint (of arbitrary degree), or as a unary dynamic constraint.

```
visibleVariable goalScored() /*only one*/
/*yes,no*/
{
      // only two possibilities
      double * returnValue = (double*)malloc(2*sizeof(double));

      if(goal == 0) // return no
      {
            returnValue[0] = 0.0;
```

```
            returnValue[1] = 1.0;
      }
      else // return yes
      {
            returnValue[0] = 1.0;
            returnValue[1] = 0.0;
      }
      return returnValue;
}
```

Elsewhere in the C++ code the *goalScored* function is assigned function number 13 as shown below.

```
      tableCallFunction[13] = goalScored;
```

The dynamic constraint is set at run time using the following function with the parameter 13, which calls the *goalScored* function.

```
double * getTablePreferences(int requestedTable)
{
      double * temp = (*tableCallFunction[requestedTable])();
      return temp;
}
```

Functions that have a number appended on their name by the java GUI to break name dependencies call the same function to set the dynamic constraint. So *seeScorer01*, *seeScorer02*, and *seeScorer03* all call the *seeScorer* function to set the dynamic constraint.

Appendix B

S Satisfies Properties of a Semiring

$$S \quad = \quad \langle [0,1], f_+, f_\times, \mathbf{0}, \mathbf{1} \rangle$$

where f_+ is implemented using *max*, and $f_\times$ is implented using arithmetic multiplication.

- $\mathbf{A} = [0,1]$ is a set with $\mathbf{0}, \mathbf{1} \in \mathbf{A}$.
- The + operator, *max* is commutative

$$\begin{aligned} max(a,b) &= max(b,a) \\ &= \begin{cases} a & \text{if } a \geq b \\ b & \text{if } b > a \end{cases} \end{aligned}$$

- The + operator, *max* is associative

$$\begin{aligned} max(a, max(b,c)) &= max(max(a,b), c) \\ &= \begin{cases} a & \text{if } a \geq b \text{ and } a \geq c \\ b & \text{if } b > a \text{ and } b \geq c \\ c & \text{if } c > a \text{ and } c > b \end{cases} \end{aligned}$$

- $\mathbf{0}$ is the unit element for the + operator, *max*

$$max(\mathbf{0}, a) = a = max(a, \mathbf{0})$$

- The $\times$ operator, implemented with arithmetic multiplication, is associative with $\mathbf{1}$ as its unit element and $\mathbf{0}$ as its absorbing element, since arithmetic multiplication has these properties.
- The $\times$ operator distributes over *max*. i.e., for any $a, b, c \in \mathbf{A}$

$$\begin{aligned} a \times max(b,c) &= max((a \times b), (a \times c)) \\ &= \begin{cases} a \times b & \text{if } b \geq c \\ a \times c & \text{if } c > b \end{cases} \end{aligned}$$

Appendix C

Graphs used to Support Modeling using Linear Regression

Shown here are graphs generated using a model where the time required to select a camera is a function of the number of cameras, raised to the power of the number of displays. Under this model each graph should appear roughly linear, since the data is first transformed (linear, square root, cube root, or fourth root as determined by the number of displays). Graphs of experiments using only one display are not shown in this appendix, since the graph is already linear. The transformations shown here were done using Microsoft Excel, so there may be minor variations in the transformations compared to the regression analysis performed by R.

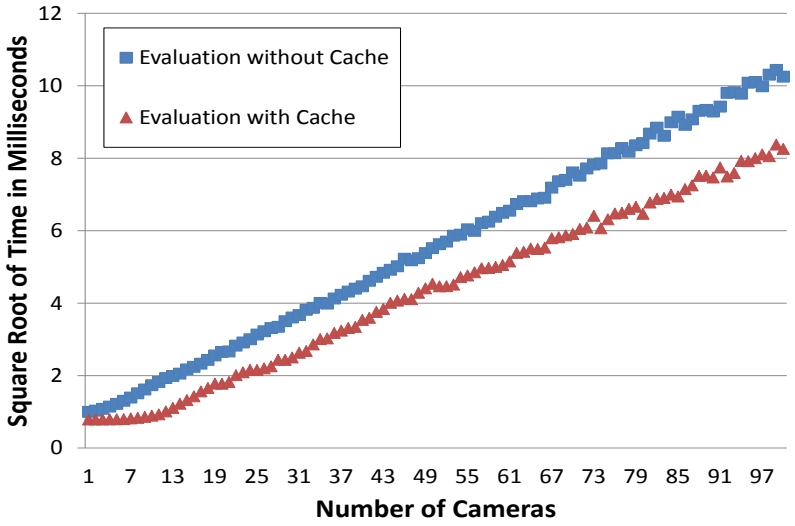

Figure C.1: Cache vs. No Cache with Two Displays, constraints *KeepCentered* and *DistanceToCamera*

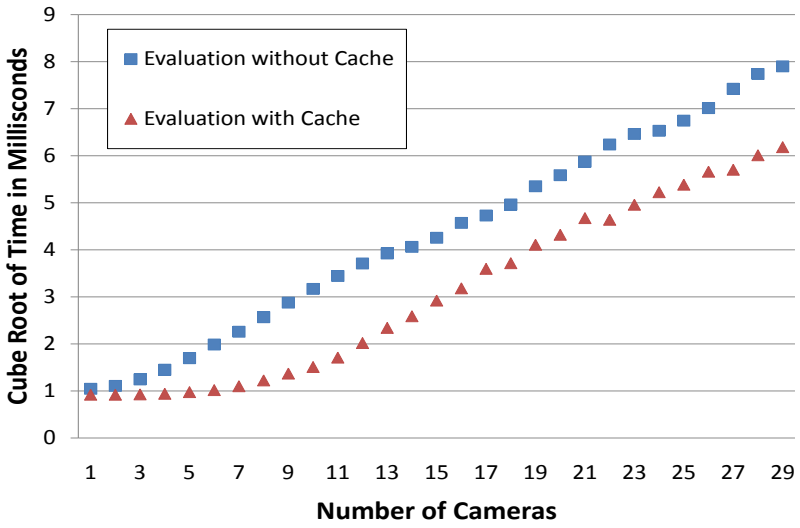

Figure C.2: Cache vs. No Cache with Three Displays, constraints *KeepCentered* and *DistanceToCamera*

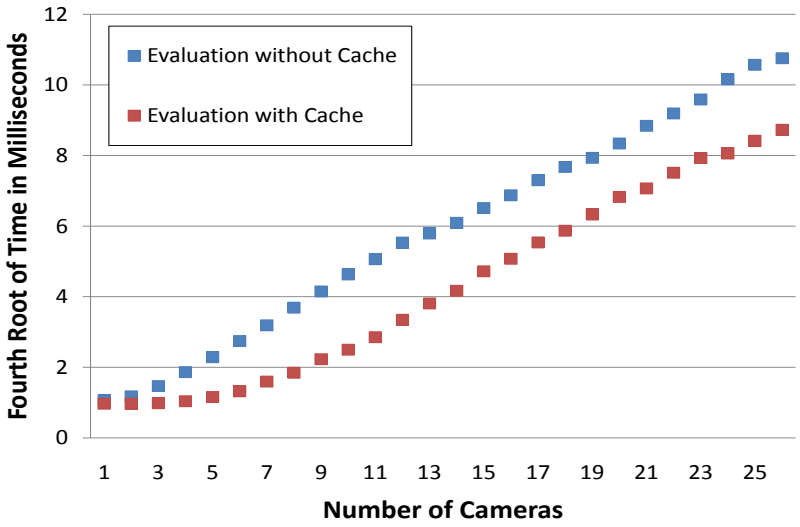

Figure C.3: Cache vs. No Cache with Four Displays, constraints *KeepCentered* and *DistanceToCamera*

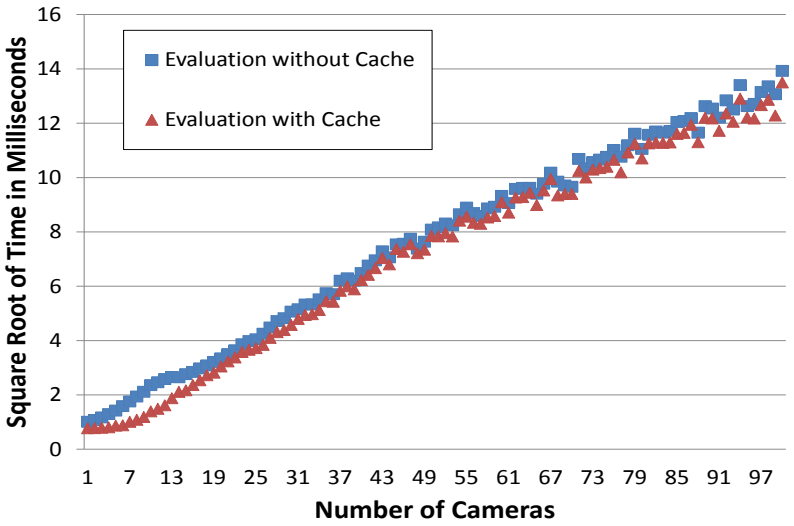

Figure C.4: Cache vs. No Cache with Two Displays, constraints *KeepCentered*, *DistanceToCamera*, and *FrameCoherence*

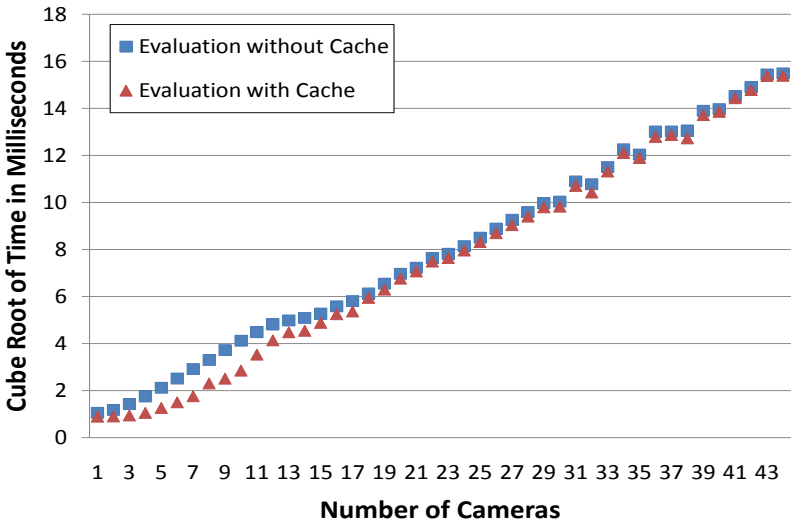

Figure C.5: Cache vs. No Cache with Three Displays, constraints *KeepCentered*, *DistanceToCamera*, and *FrameCoherence*

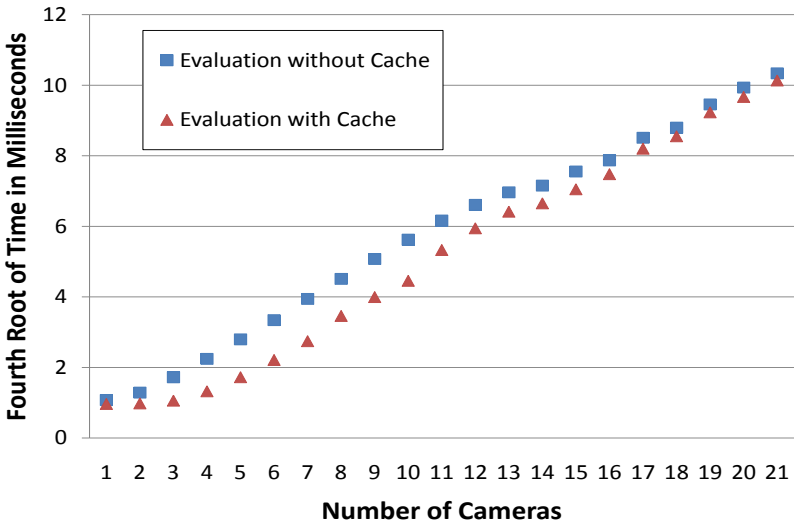

Figure C.6: Cache vs. No Cache with Four Displays, constraints *KeepCentered*, *DistanceToCamera*, and *FrameCoherence*

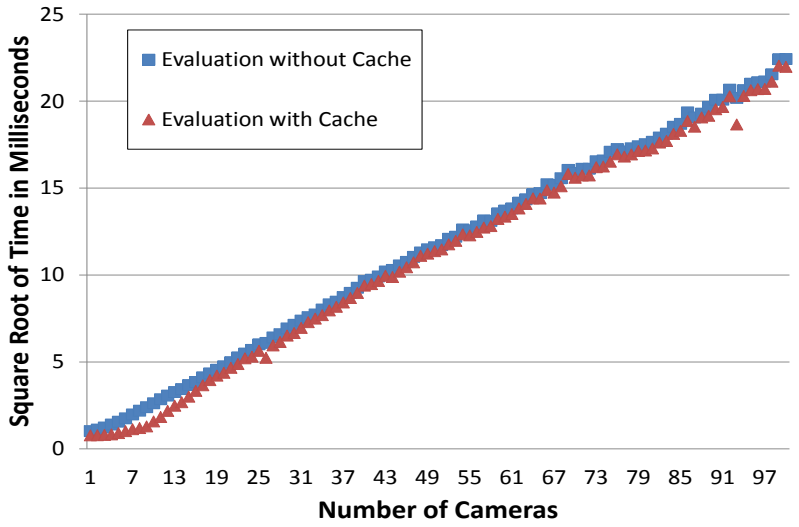

Figure C.7: Cache vs. No Cache with Two Displays, constraints *KeepCentered*, *DistanceToCamera*, *FrameCoherence*, and *GoalScored*

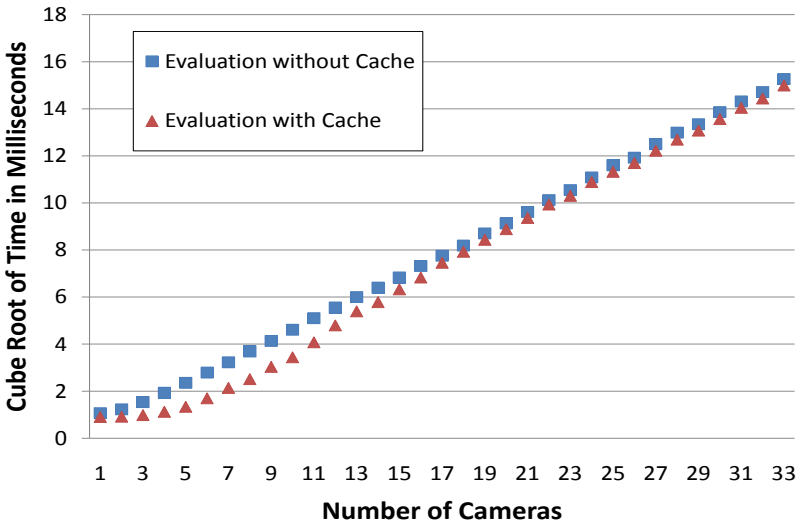

Figure C.8: Cache vs. No Cache with Three Displays, constraints *KeepCentered*, *DistanceToCamera*, *FrameCoherence*, and *GoalScored*

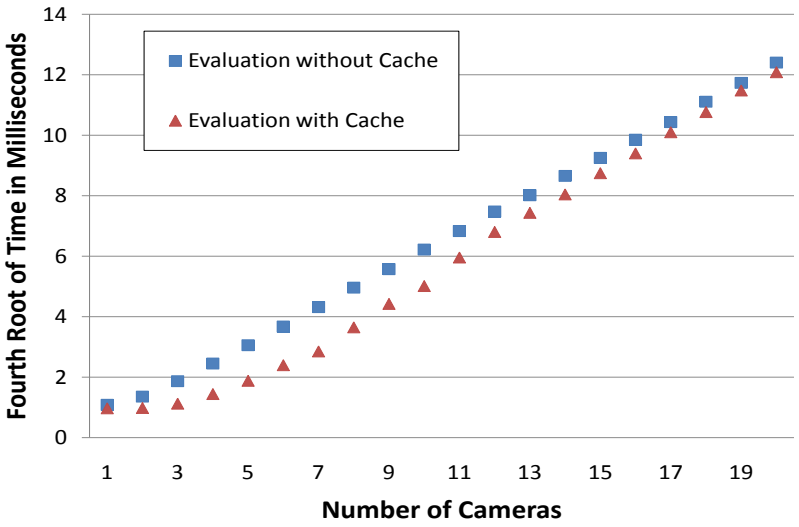

Figure C.9: Cache vs. No Cache with Four Displays, constraints *KeepCentered*, *DistanceToCamera*, *FrameCoherence*, and *GoalScored*