Proceedings of the Twenty-Fourth International Florida Artificial Intelligence Research Society Conference

A Method of Virtual Camera Selection Using Soft Constraints

Michael Janzen and **Michael Horsch** and **Eric Neufeld**
mike.janzen@usask.ca, eric@cs.usask.ca, horsch@cs.usask.ca
Department of Computer Science
University of Saskatchewan, Canada

Abstract

We describe a software tool to select among camera feeds from multiple virtual cameras in a virtual environment using semiring constraint satisfaction problem techniques (SCSP), a soft constraint approach. We show how to encode a designer's preferences, and select 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 (ie. no prediction). To reduce computation costs for a final implementation, the SCSP evaluation can be cached and converted to native code. Our 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 (constraint satisfaction problems) are made.

Introduction

Most video games, movies, and television shows use a dynamic camera viewpoint that varies the user's viewing experience. Previous automated virtual camera work uses constraint systems to place or move the camera, but little treatment has been given to the problem of selecting among multiple virtual cameras.

Systems that automatically place and move a virtual camera, which could supply the camera feeds for selection, are discussed in the next section. Given infinite resources, camera placement and feed selection become equivalent. From the camera placement perspective, the camera parameters come from an infinite number of locations and angles. From the camera selection perspective, the system selects from an infinite number of cameras that span every location and angle. However, given limited resources, camera placement algorithms prioritize searching nearby locations, where camera selection considers a finite number of discrete locations.

We encode a camera director's preferences for selecting a camera feed into a collection of preference functions in a modular fashion using a soft constraint representation. We emphasize dynamic scenes such as sports games and third person shooters, where action is not entirely predictable (as opposed to a scripted scene or a conversation between two or three people). We demonstrate a real-time software tool for the selection process using a spectator viewpoint virtual hockey game and a third person perspective maze game. Multiple screens provide a convenient way to increase the complexity of the constraints.

In the sequel, we describe the semiring constraint satisfaction problem (SCSP) formalism, and ways to express a sequence of preferences that progressively refine the visual experience. As we developed these preferences in the field, we found trade-offs among preferences.

Virtual Camera Placement

Automated camera placement methods can be generally grouped into two approaches: through-the-lens systems and constraint systems. For a comprehensive survey see work done by Christie and colleagues (2008).

In a through-the-lens system a designer specifies points in world space to pin to points in screen space (Gleicher and Witkin 1992; Christie and Hosobe 2006). As the scene evolves the camera placement system adjusts the location, angle, and zoom level to keep the points in world space at the same screen location. Designers avoid over-constrained cases, or the error is distributed in a least squares fashion.

Camera placement using pure constraint systems require the designer to specify constraints for the system to manage, such as height, camera angle, and an unoccluded view of an object or subject (Drucker and Zeltzer 1994; Bares and Lester 1999). The system selects camera parameters to satisfy all of the constraints. When the problem is over-constrained, less important conflicting constraints are dropped until all remaining constraints are satisfied (Bares and Lester 1999). A frame coherence constraint can cause the camera to select nearby locations and angles (Halper, Helbing, and Strothotte 2001).

Bourne presents a camera control system using soft constraints (2008). The camera placement method searches nearby locations using a sliding octree solver, which searches nearby locations in a divide and conquer fashion from a coarse to fine resolution. Each potential location is evaluated using a weighted constraint satisfaction problem (CSP) in which weights are assigned to each constraint. A potential location is assigned a score based on the sum of the weights of the constraints the location satisfies. This approach differs from dropping less important constraints, in that multiple, less important constraints may be satisfied at the expense of violating a more important constraint.

Virtual Camera Selection

He *et al*. introduce a mechanism to cut between cameras based on film conventions (1996). Idioms, i.e. knowledge of camera cuts, are encoded into finite state machines. States encode parameters for the current camera, while transitions between states represent cuts between cameras. For example, an idiom for filming a two person conversation may start with an establishing shot and then alternate between the speaker and listener. Bhatt and Flanagan discuss camera selection in scripted scenes using event calculus (2010).

Idioms are less suited to dynamic scenes, where the flow of action is less predictable, than scripted scenes (Turkay, Koc, and Balcisoy 2009; Bourne, Sattar, and Goodwin 2008). To change to any camera, an idiom approach would require a completely connected finite state machine. Guards on transitions (conditions to satisfy before changing states) are comparable to hard constraints; the system may remain with a sub-optimal camera choice since the guard is not completely satisfied. Similarly, although not stated, the system chooses randomly if more than one guard is satisfied.

Assa *et al*. present a camera selection method based on correlation with motion capture data (2010). The pixels from each potential camera are correlated with the motion capture data, and the view with most motion in the scene is displayed. Similarly Passos *et al*. use a neural network to select between three camera feeds in a racing game (2009). Their system learns from user input, rather than allow a designer to specify important conditions for selecting a camera feed. Our system differs from both of these systems by enabling a designer to explicitly specify important conditions to capture on camera.

SCSP Formalism

The problem is to find a set of assignments for a set of variables related by a set of constraints. For in depth treatment of constraint satisfaction problems (CSP) see (Dechter 2003). A semiring constraint satisfaction problem (SCSP) is a generalization of CSPs and CSP-like problems. Both classical CSPs and partial CSPs (PCSP), such as a weighted CSP, can be represented in the SCSP framework. A key difference between a classical CSP or PCSP is a SCSP constraint may be satisfied to a degree. A PCSP can be partially satisfied if some of its constraints are fully satisfied and others are not. The key idea in the generalization of SCSPs is to represent constraints as functions rather than relations. This requires that the function return values that combine and compare in a few restricted ways. When preferences in a SCSP are specified such that there is a total order (i.e. all values are comparable) a join operation enables a ranking over the global preference (Freuder and Wallace 1992). A solution to a SCSP is a tuple that has the maximum global preference.

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, $\mathbf{X}$ is a set of variables, and $\mathbf{D}$ is a set of corresponding domains, which are typically discrete and finite. $\mathbf{A}$ is a set of preference values including $\mathbf{0}$ and $\mathbf{1}$, which represent the minimal and maximal values respectively. The operators $+$ and $\times$ operate over pairs of elements of $\mathbf{A}$ and are closed, commutative, and associative. Their definition changes depending on which SCSP implementation is chosen (we use *max* and multiplication for our work).

Each constraint in $\mathbf{C}$ is a function mapping outcomes of variables included in the constraint to preference values; thus the constraint defines a function over some of the variables in $\mathbf{X}$. Functions are joined using a join operator, denoted $\otimes$, which uses $\times$ to combine preferences. The combination of all constraints 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 $\mathbf{X}$; specifying independent constraints in multiple, smaller tables is more tractable. Typically, for implementation a branch and bound method searches the global preference. As the search descends the tree, constraints involving set variables are applied to the partial solution. Our implementation uses a best first branch and bound search to first traverse branches with a higher scored partial assignment. For more information on SCSPs see (Bistarelli, Montanari, and Rossi 1997; Bistarelli et al. 1999).

Design

We introduce a camera selection mechanism based on a semiring, $\langle \mathbf{A}, +, \times, \mathbf{0}, \mathbf{1} \rangle$. Here $\mathbf{A}$ is the closed unit interval, with $\mathbf{0}$ and $\mathbf{1}$ equal to their numerical values respectively. We implement $+$ with *max* and $\times$ with multiplication. It is trivial to show that these choices satisfy the properties of a semiring. $\mathbf{X}$, $\mathbf{D}$, and $\mathbf{C}$, are dependent on the implementation; a hockey example is presented below.

A designer's camera preferences are encoded in constraints, $\mathbf{C}$, which we represent with tables. For example, in our hockey game, one constraint is *KeepCentered* specifying that the hockey puck appear in the center of the camera. Another constraint is *DistanceToCamera*, which specifies that the puck appear closer to the camera rather than further away. These are just one viewer's preferences; another viewer may have different preferences. These constraints can be encoded in tables, shown in Table 1 using variables location and distance. Intuitively higher values indicate greater preference. Any preference function over a finite set of discrete variables can be encoded in a similar fashion. Table 1 shows unary preferences (one variable), but preferences can be defined over more variables in general.

Location	*Pref*
center	1.0
border	0.7
out	0.1

Distance	*Pref*
near	1.0
middle	0.8
far	0.1

Table 1: *KeepCentered* and *DistanceToCamera* preferences

Here the variables are $\mathbf{X}$ = $\{$*Location*, *Distance*$\}$. Preferences among variable outcomes in one constraint are expressed independently of preferences of variable outcomes in another constraint. Interdependencies between variables arise in more complex situations, and can be easily modeled.

Since the join operator is based on arithmetic multiplications, when the two constraints are joined, the best tuple in the global preference is $\{Location = center, Distance = near\}$, with a preference of 1.0.

To know which camera to select, a *Feed* variable must be introduced. For example, the constraint for two cameras, both equally preferred, is shown in Table 2. If cameras are not equally preferred, such as in the maze game described later, then the preference values would reflect this.

Feed	*Pref*
one	1.0
two	1.0

Table 2: A Preference over Camera Feeds

Static Constraints versus Dynamic Constraints

Static constraints are set at design time, and *dynamic* constraints are set automatically during execution. The constraints shown in Table 1 are static constraints, and may not reflect the current situation. It may be that no camera has the puck centered in the camera view, with a near distance. For example, the dynamic constraint shown in Table 3 provides evidence that the puck is in the border of camera one's view. Dynamic constraints provide evidence by assigning a preference level of **0** to tuples inconsistent with the current state of the virtual environment.

Feed	*Location*	*Pref*	*Feed*	*Location*	*Pref*
one	*center*	0.0	*two*	*center*	0.0
one	*border*	1.0	*two*	*border*	0.0
one	*out*	0.0	*two*	*out*	1.0

Table 3: Example Dynamic Preference Table

Considering the static and dynamic constraints from Tables 1, 2 and 3, the best tuple from the global preference is now $\{Feed = one, Location = border, Distance = near\}$, with a preference of 0.7. Similarly, adding a dynamic constraint for the distance ensures that only tuples with a distance corresponding to reality are selected as the best tuple. Dynamic constraints are set just before the system selects a feed.

Additional Constraints

We represented a variety of additional constraints using this formalism. A unary *BiasCamera* constraint prefers some camera viewpoints to others. In the maze game this constraint allows the system to prefer viewpoints behind the avatar to those in front, making the avatar easier to control. The constraint is similar to Table 2, but has 33 domain values for *Feed* with some values less than one.

A temporal constraint, *FrameCoherence* [1], prevents rapid changes between camera feeds. Table 4 shows a designer's preference to prefer feeds currently used for less than five seconds versus camera feeds selected for more than ten seconds. The last two values in Table 4 apply to cameras that are not currently selected.

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 4: Preference for Frame Coherence

A potentially unwanted viewing effect occurs when using *FrameCoherence* and multiple screens. A camera feed selected for less than two seconds has a high preference, but the preference does not specify on which screen the feed should be displayed. Thus the feed could rapidly switch between different screens. To avoid this, we introduce a *wasFeed* variable for each screen. 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 screen. When used with fixed position cameras *wasFeed* 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 (Haigh-Hutchinson 2009). The preferences we use are non-trivial; a constraint graph using 4 screens, 29 variables and 59 constraints is shown in Figure 1. The figure contains

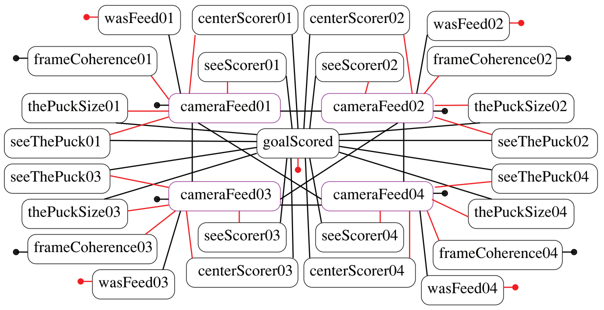


Figure 1: Constraint Graph Example using Four Displays

constraints to show the scorer after a goal has been scored, which are similar to preferences to show the puck. The numbers on the ends of variables are used to break name dependencies. For example, without a variable for each display, specifying a near distance on screen one would also force a near distance preference on screens two, three and four. The lines terminated in a dot represent unary constraints.

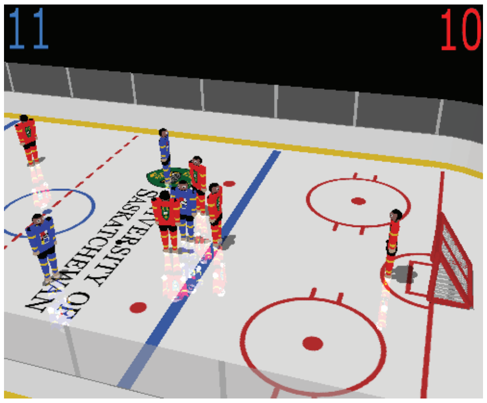
Occlusion constraints keep particular characters or objects in view. In the maze game the player searches for targets, so views with a target are preferred to those without. However, the target may be occluded by a wall. A constraint over the variable $TargetVisible \in \{well, some, poor\}$ encodes the preference to see a target. To set the dynamic constraint,

[1]We call this constraint *FrameCoherence* since its function is to maintain camera frame coherence, similar to the frame coherence objective in (Halper, Helbing, and Strothotte 2001).

the player is bounded by a non-visible box, which is divided into 27 smaller boxes ($3 \times 3 \times 3$). A count of the number of these smaller boxes that are visible determines the dynamic constraint assignment to *TargetVisible*.

Problem Domain

We constructed two virtual environments: a spectator perspective hockey game and a third person maze game, shown in Figures 2a and 2b. In the hockey game, with one display and ten cameras our SCSP search completes in approximately one millisecond, but displays acceptable results when run every 30 frames (one evaluation per second).



(a) Hockey Game

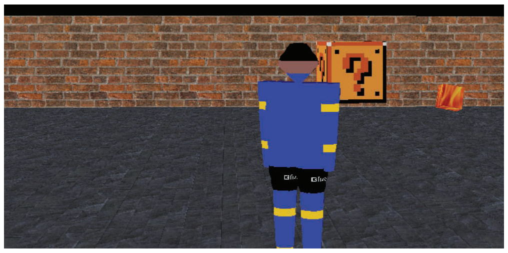

(b) Maze Game - player firing a box (right) at a target box

Figure 2: Hockey and Maze Games

In the hockey game, we wanted to keep the puck in the center of view as much as possible, while keeping the action as close as possible. Other viewers may have other preferences; we choose these as a baseline. An independent simulator controls players and ten cameras are placed around one side of the rink. The game can implement constraints such as those presented in Table 1 or Figure 1, but considers ten cameras instead of two.

For the dynamic constraint *KeepCentered* the screen is considered to range from -1 to $+1$ (ie. normalized coordinates). The puck is considered to be in the middle of a camera's view if its x and y coordinates lie between -0.4 and 0.4, border of the screen between -1 and 1 (but not in the middle), and out of view otherwise. The dynamic constraint for *DistanceToCamera* is similarly encoded, but based on the z coordinate of the puck.

In the maze game, the goals of the camera selection system are to choose a view that helps the player see the avatar, and provide cinematic cues for objectives in the game. The user controls an avatar in a 3D maze and fires boxes at target boxes. The player wins when all target boxes have been hit. Camera view constraints include keeping the avatar in view (*AvatarVisible*), seeing a firing box (*FiringBoxVisible*), seeing a target box (*TargetBoxVisible*), and being behind the avatar (*BiasCamera*). A *VisibleTransition* constraint avoids camera changes that pass through walls. Smooth camera changes are performed by linearly interpolating between camera views, rather than having abrupt camera feed changes, as we chose with the hockey game. This is similar to commercial games that use a third person perspective.

We implement occlusion constraints by bounding objects using rectangular cuboids and projecting these to rectangles on a 2D screen. For example, the player's avatar is divided into 27 boxes ($3 \times 3 \times 3$) and the visibility is classified into *well* if less than 13 boxes are occluded, *some* if less than 18 boxes are occluded, or *poor* otherwise. Once on the 2D screen a rectangle is occluded if it is overlapped by a rectangle from another object that has a lower z value (ie. closer). The *VisibleTransition* constraint uses the occlusion information of the cameras along potential camera paths to determine the degree of occlusion of the considered camera path.

Cache Acceleration

As time goes on in the simulation, configurations reoccur whenever dynamic constraints for the current evaluation are identical to a previous evaluation. Consequently we implemented a cache to take advantage of the temporal locality and reduce the average evaluation time. Figure 3 plots the average time to select a camera based on the number of cameras available. Each data point in Figure 3 is the average of 1000 evaluations using the hockey game with four displays and the preferences *KeepCentered* and *DistanceToCamera* from Table 1. The cache is implemented using a hash table with arrays for items mapped to the same hash value.

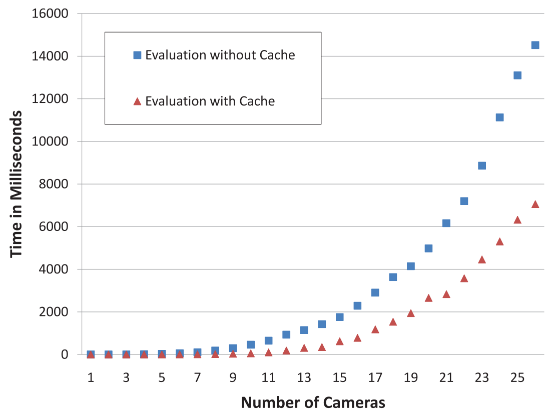


Figure 3: Computation Time Cache vs. No Cache

The triangles in Figure 3 show that the cache improves the average camera selection time, but has noticeable lag with a large number of cameras. However the cache technique is still 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.

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, which execute more quickly than searching for the best feed selection. However, this approach has the same software maintenance problem that large expert systems did, namely that large rule bases are difficult to modify or extend as the number of rules increase (Jackson 1986).

An advantage of our 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, our system 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 the preferences in a small table. With native code, changing one if–statement can have unexpected interactions.

Conversion To Native Code

To achieve the potential benefits of both approaches we demonstrate how to automatically convert the SCSP system to native code. Thus the camera selection system is designed under the SCSP scheme, but 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 (3^{20} considering only *KeepCentered* and *DistanceToCamera* with ten cameras and one display). Adding additional constraints increases the number of combinations, and thus evaluation time, with exponential complexity assuming the new constraints contain new variables.

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. We generate likely dynamic constraint combinations by sampling game execution, and store them in a cache large enough to avoid conflicts. The samples form a training set from which a decision tree is generated using an information gain heuristic. The decision tree is written to C code if-statements, which are compiled to native code. When the native code is provided with dynamic constraints that were not in the sampling, it returns the most frequently used camera in the decision tree's subtree.

The quality of native code camera selection approaches the SCSP system asymptotically. Consequently, a high performance can be achieved in a relatively short time as shown in Figure 4 which uses the *KeepCentered* and *DistanceToCamera* preferences and one screen. With large SCSP systems the collection of cache samples, and conversion to native code, can be parallelized to reduce conversion time.

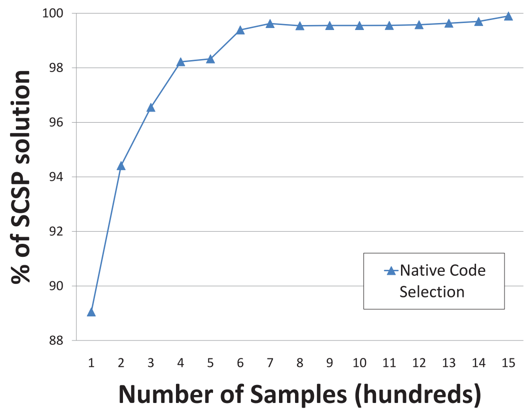


Figure 4: Native Code Performance versus SCSP solution

Findings

In the hockey game, using *KeepCentered* and *DistanceToCamera* our camera selection system effectively chooses views that center the puck with a near view when available, and avoids far views with the puck out of view. When the preferences values are reversed, the system appropriately selects far views with the puck out of view. Initially, we implemented only the *KeepCentered* constraint, but found that the camera selection system often selected views from the opposite end of the hockey rink, if the puck was centered. Adding the *DistanceToCamera* constraint required the system to also consider the distance to the puck, resulting in what we feel is a more pleasing camera selection. Adding the *FrameCoherence* constraint prevented the system from switching between camera feeds too quickly (less than two seconds), or remaining on one feed for too long.

In the maze game, we first implented the *BiasCamera* and *AvatarVisible* constraints; the game was playable, but we wanted to encourage seeing target boxes, as this is the goal of the game. We then added the *TargetBoxVisible* and *FiringBoxVisible*, but found the camera passing through walls disorienting. Increasing the preference to see the player solved this problem, along with adding the *VisibleTransition* constraint to avoid camera changes that pass through walls.

Evaluation

Camera placement methods have used CSPs with hard constraints, and He *et. al*'s Virtual Cinematographer (He, Cohen, and Salesin 1996) uses hard constraints in that guards on transitions between states are satisfied or unsatisfied. Since there are no implementations of these systems available, we examine how our soft constraint system (preferences) compares to a hard constraint system (CSP), and mapped our preferences to CSP constraints. For the values in Tables 1 and 4 we threshold values 0.5 and greater to 1 and values less than 0.5 to 0. Other mappings are possible. For example the designer 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 *DistanceToCamera* for at least 10000 feed selections from 10 cameras. Additional constraints preferred different camera feeds on each screen. The tests were repeated with an added *FrameCoherence* constraint. As Table 5 shows the CSP method frequently finds solutions when the problem is under-constrained, but finds solutions less often as the problem becomes over-constrained resulting in finding no solutions with three constraints and five displays. Our SCSP solution was able to

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 5: Percent of Samples CSP solved

find a solution 100% of the time, albeit by partially satisfying constraints as the system became over-constrained. Notice 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.

Discussion

Our approach offers designers a tool to select cameras at run time, which may result in better camera selection in games and other virtual environments, faster development of the camera selection system with an easier method of modification, and potentially allow users (players) to customize the camera selection system after a games release. Our system is flexible in that it can handle different types of constraints, such as centering objects and avoiding occlusion, and 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. By design our system expands to handle multiple screens. For improvement in run-time performance our system can generate native code, which still allows a camera selection design in modular tables.

Acknowledgements

The first author thanks NSERC and the University of Saskatchewan for support in the form of a PGS-D and departmental scholarship. The research of the second and third authors is supported by NSERC Discovery Grants.

References

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

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

Bhatt, M., and Flanagan, G. 2010. Spatio-Temporal Abduction for Scenario and Narrative Completion (a preliminary statement). *International Workshop on Spatio-Temporal Dynamics*.

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

Bistarelli, S.; Montanari, U.; and Rossi, F. 1997. Semiring-based constraint satisfaction and optimization. *Journal of the ACM* 44:201–236.

Bourne, O.; Sattar, A.; and Goodwin, S. 2008. A Constraint-Based Autonomous 3D Camera System. *Constraints* 13(1-2):180–205.

Christie, M., and Hosobe, H. 2006. Through the Lens Cinematography. In *Proceedings of the 6th International Symposium on Smart Graphics*.

Christie, M.; Olivier, P.; and Normand, J.-M. 2008. Camera Control in Computer Graphics. *Comput. Graph. Forum* 27(8):2197–2218.

Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann Publishers.

Drucker, S. M., and Zeltzer, D. 1994. Intelligent Camera Control in a Virtual Environment. In *Proceedings of Graphics Interface '94*, 190–199.

Freuder, E. C., and Wallace, R. J. 1992. Partial Constraint Satisfaction. *Artif. Intell.* 58(1-3):21–70.

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

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

Halper, N.; Helbing, R.; and Strothotte, T. 2001. A Camera Engine for Computer Games: Managing the Trade-Off Between Constraint Satisfaction and Frame Coherence. In Chalmers, A., and Rhyne, T.-M., eds., *EG 2001*, volume 20(3). Blackwell Publishing. 174–183.

He, L.; Cohen, M. F.; and Salesin, D. H. 1996. The Virtual Cinematographer: A Paradigm for Automatic Real-Time Camera Control and Directing. *Computer Graphics* 30(Annual Conference Series):217–224.

Jackson, P. 1986. *Introduction to Expert Systems*. Addison-Wesley.

Passos, E. B.; Montenegro, A.; Clua, E. W. G.; Pozzer, C.; and Azevedo, V. 2009. Neuronal editor agent for scene cutting in game cinematography. *Comput. Entertain.* 7(4):1–17.

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