

A Constraint-Based Autonomous 3D Camera System

Owen Bourne, Abdul Sattar
Institute for Integrated and Intelligent Systems
Griffith University
Brisbane, QLD 4111
research@owenbourne.id.au, a.sattar@griffith.edu.au

Scott Goodwin
School of Computer Science
University of Windsor
Windsor, ON N9B 3P4
sgoodwin@uwindsor.ca

August 3, 2007

Abstract

Camera control techniques for Interactive Digital Entertainment (IDE) are reaching their limits in terms of capabilities. To enable future growth, new methods must be derived to address these new challenges. Existing academic research into camera control is typically devoted to cinematography and guided exploration tasks, and is not directly applicable to IDE. This paper describes a novel application of constraint satisfaction in the design of a camera system that addresses the unique and difficult challenges of IDE. It demonstrates a specialized constraint solver that exploits the spatial structure of the problem, enabling the real-time use of the camera system. The merit of our solution is highlighted by demonstrating the computational efficiency and ability to extend the cameras capabilities in a simple and effective manner.

1 Introduction

The primary role of the camera in Interactive Digital Entertainment (IDE) is to provide a window between the user and the environment. This window defines not only how and what the user sees, but also the way in which the user interacts with elements of the environment. Limitations of existing commercial camera systems are well known. Every gamer has had the frustrating experience of missing a crucial jump or getting pummeled by enemies because the camera became obstructed or faced the wrong way. As such, camera control plays a pivotal role in how the user perceives the environment and the effectiveness of the experience. It is the responsibility of the camera control system to calculate the appropriate camera properties to present the user with their view of the virtual environment for each frame of animation.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

Achieving competence in a camera control system is a complex task which requires management of the trade-offs between system and development complexity, and the overall effectiveness of the camera. Camera systems for virtual environments introduce new problems not apparent in situated environments, as the camera movement, focal length, and field-of-view are not bound by physical laws.

Without the influence and restrictions of real-world constraints on the camera properties, achieving smooth movement in a virtual environment is difficult. In addition to smooth movement, the camera control system is responsible for positioning the camera appropriately to setup the viewing properties of the scene. In the simplest cases, these viewing properties can be easily defined and evaluated. However, as the viewing properties become more complex (additional targets, denser and more dynamic environments, complex framing properties), autonomously controlling a virtual camera becomes increasingly difficult.

The task of controlling the virtual camera in a realistic and coherent manner is the core problem of autonomous camera control for IDE. What appears on the surface to be a straightforward problem is compounded by the nature of the problem domain. Visualizing multiple fast-moving targets, avoiding occlusions in dense and highly dynamic environments, and handling unpredictable user control of targets are a few of the often conflicting issues that must be resolved. The real-time nature of IDE makes the efficiency of the solution very important.

While the theory of camera control is useful in domains outside of IDE, the strict requirements of IDE provide an ideal testbed for development. Not only do IDE environments provide a low-cost alternative to real-world testing, the ever increasing complexity and realism of the environments makes them a plausible domain for evaluating all manner of new technologies [28].

1.1 Motivation

This paper is practically oriented, and as such the research is driven by deficiencies in the camera control methods used within the IDE industry. Current camera control methods used by industry for IDE are very application specific, often requiring many special case fixes. The camera system often needs to be re-written or re-designed for use in another application, often requiring the re-tuning of many parameters. This reduces code re-use, and adds time and cost to IDE development.

The requirements for an autonomous camera system for IDE are:

- **Real-time.** The camera must be responsive to user input and operate under interactive time constraints.
- **Competent.** The camera must maintain a clear view of the currently selected character(s) and provide coherent motion to maintain this view.
- **Extendable.** The camera must use a flexible architecture to adapt to changing demands required by games.
- **Dynamic.** The camera must deal with complex and dynamic environments.
- **Environment independence.** The camera must be able to deal with a variety of 3D worlds commonly found in games.
- **Ease of implementation.** The camera must not require pre-processing of the environment.

- **Cinematic.** The camera must support cinematic functionality.

Industry based methods are computationally efficient, however they lack the ability to express complex visual properties such as the spatial configuration of objects in the rendered image. As IDE applications become increasingly complex, so do the demands on the camera system. Existing methods often have little or no potential for expanding their capabilities, and are therefore of limited use for future applications.

Existing research from academia is primarily focused on scientific visualization and automated cinematography. Neither of these domains are directly comparable to IDE, as they have differing computational and visualization requirements. The research is often more concerned with the high-level camera specification than the low-level motion of the camera.

As a result of this focus, existing academic methods only partially address the issues associated with IDE camera systems. They often provide frameworks that are very general in nature, and do not translate effectively to the specific requirements of IDE camera systems. There is often no exploitation of problem-specific optimizations, and a reliance on very general solution methods. This makes the computational performance prohibitive for applications in the IDE domain.

With the growing importance of competent camera systems for interactive games, research to address the IDE-specific problems remains an open issue. There is currently a lack of camera systems that address the increasing need for flexible and cinematic capabilities for IDE. Constraint satisfaction methods provide a good framework for addressing these design challenges.

The constraint satisfaction field has a rich and varied research history, providing a rich representation for many problem spaces [27]. Highly specialized algorithms have been created to solve problems represented using this framework. By leveraging this research, considerable benefits in the design and implementation of an autonomous camera system for IDE can be achieved.

This paper is an extension of our original work in [9], and is organized as follows: the next section covers the necessary related work in the field. The weighted-CSP framework used by our camera system is described in Section 3, with a detailed discussion of the constraint solving process described in Section 4. Extensions to the camera system are covered in Section 5, followed by the conclusion and directions for future research.

2 Related Work

Camera control methods used in industry are often based on efficient mathematical techniques. Methods such as polar coordinates [17] and spherical coordinates [31] are common. To address movement issues that arise with these approaches, spring systems are often applied to produce more *natural* camera motion [26].

Some recent work has investigated the use of more sophisticated camera control methodologies for IDE. Quaternion interpolation [7] and proportional controllers [21] provide in-built mechanisms to generate smooth camera motion. The use of steering behaviours applied to a virtual camera have also been investigated [12].

2.1 An Overview of Constraints

The most common representation for constraint-based camera systems is defined here to reduce repetition, and where appropriate, variations of this representation are de-

Constraint	Description
Size	The size of an object in the rendered image, usually specified as a percentage of the frame area.
Placement	The placement of an object in the rendered image, usually specified as left, right or centre.
Vantage Angle	The altitude to view the object from, sometimes represented as height.
Viewing Angle	The angle to view the object from, usually determined with respect to the facing direction of the object.
In View	To ensure that a given object is within the camera view.
Exclude	To ensure that a given object is not within the camera view.
Depth Order	Specific depth ordering of two or more objects, where one must be closer to the camera than the other.
Occlusion	Minimize (or avoid) the amount of occlusion of an object.

Table 1: Common Constraints.

scribed. The variables of the camera control problem typically map directly to the camera position (X, Y, and Z are treated as three distinct variables), camera viewpoint (X, Y, and Z), and the camera’s Field-Of-View (FOV), totaling seven variables.

The domains for each of the constraints differs, however for the camera position and viewpoint variables the domain is typically a subset of possible world coordinate positions that the camera can occupy. For the FOV variable a set of values between the two extremes of the FOV are used, often mapped between 30° and 100°.

The constraints dictate the visual preferences of the resulting image, and can easily represent complex visual properties. The current instantiation of values to the variables is compared against the desired camera configuration (constraints) to find a solution. A non-exhaustive list of common constraints and what they do is shown in Table 1. Although the names and actual implementation of constraints often varies between approaches, many of them share the same principles and constraints.

2.2 An Overview of Constraint-Based Approaches

Drucker and Zeltzer investigated the use of camera modules to control the viewing and movement properties of a virtual camera [18, 20]. Each camera module consists of a set of constraints that are necessary to achieve the desired camera behaviours, using constraints such as path planning. The goal of the camera system is to provide an optimal path through a virtual museum, with the camera path passing by any objects of interest specified by a user.

The constraints are used to align the camera to the world up vector, maintain a camera height relative to the floor, maintain the camera view normal towards the objects of interest, and maintain the camera position on a collision free path. The path through the environment is derived using an A* search algorithm. The use of a path planning algorithm and associated collision avoidance strategy makes it difficult to apply the technique in dynamic environments, as the path must be recalculated for each change in the environment. This technique is primarily used for cinematics and guided exploration purposes, but is of little practical use in IDE.

The CamDroid system [19] extended upon the initial camera framework developed in [18], by allowing camera modules to be sequenced. The camera modules are linked

1
2
3
4
5 together as nodes in a graph, with the arcs representing possible transitions between
6 them. The transitions between camera modules are enacted by matching the corre-
7 sponding simulation state to the entry and exit conditions of the current and future
8 modules respectively. This system is analogous to a finite state machine.

9 The constraint solver employed by Drucker is based on the sequential quadratic
10 programming technique [20]. The problem is formulated as a sequence of quadratic
11 sub-problems leading to a solution of the constrained problem.

12 The constraints used by Drucker and Zeltzer were combined to specify how objects
13 should appear in the final rendered image, and also any animation or path following
14 for the camera. Bares et al. developed the CONSTRAINTCAM, which constrained the
15 viewing properties of each actor within a virtual environment [3]. This allows for the
16 desired viewing properties of each actor to be defined separately, with the constraint
17 solver determining the camera position to satisfy these viewing properties.

18 When a suitable position is not found, an *incompatible constraint pair graph* is
19 generated. Constraints that are considered weaker are progressively relaxed to increase
20 the chances of finding a solution. This process is repeated until a solution is found, or
21 the camera system decides it is not possible to generate a suitable solution.

22 A similar approach is adopted by Christie and Normand in [16]. Their *semantic*
23 *space partitioning* approach generates a volume that encompasses possible camera lo-
24 cations that generate qualitatively equivalent shots. Each volume is constructed based
25 on semantic tags derived from the film grammar [1]. From the intersection of these
26 volumes, a numerical solver derives the camera position, orientation and focal point
27 that best satisfies the desired framing properties.

28 Bares and Lester extended the CONSTRAINTCAM[3] solver by proposing the use
29 of *multi-shot visualizations* when the camera cannot find a suitable position through
30 constraint relaxation [4]. Once a visualization has been deemed to have no suitable
31 camera position to satisfy the constraints, the visualization is separated to provide the
32 user with several windowed views of the objects.

33 A distant overview visualization of the scene is provided as the main view, with
34 smaller insets demonstrating objects within this environment rendered separately. This
35 allows the user to view the context of the action, while still maintaining the desired
36 views of the objects. The multi-shot visualization camera properties are extracted from
37 the Multi-Shot Frame Composer, which maintains a set of potential camera positions
38 for displaying the multi-shot visualization for one to four cameras per screen.

39 Bares et al. introduced constraint weighting on the visual constraints to provide a
40 mechanism for preference of certain visual properties [5]. The system uses a modified
41 CSP representation, with the variables being defined as the position/3, view/3, fov/1,
42 and up/3 (totaling 10 variables). The degree of satisfaction of a constraint is calculated
43 to be a fractional value between 0 and 1. The satisfaction rating for a given solution
44 is computed from a weighted sum of the satisfaction values of all of the individual
45 constraints.

46 The constraint solver in this approach is a generate-and-test method [5]. The solver
47 is run over a grid of points with dimensions of 20x20x20 points, which constructs
48 regions of space to cull points that fail to satisfy camera positioning constraints. The
49 point with the highest percentage of satisfaction is used as the camera position to render
50 the current frame.

51 This solving mechanism is refined in [2], by refining the scale of the grid recur-
52 sively. The first pass of the solver evaluates the potential solutions in a 9x9x9 grid,
53 with the top five positions retained as the centre points of a more refined pass. Each
54 of these five positions then has a grid of 4x4x4 elements (with smaller dimensions), to
55
56
57

1
2
3
4
5 refine the search around the regions most likely to contain a preferable solution. This
6 process is performed recursively until a suitable solution is found, or the search has
7 exhausted its time or moves. As the number of recursions increases, the number of po-
8 tential solutions evaluated increases significantly, thereby increasing the computational
9 time required by the solver.

10 A more sophisticated approach is used by Languenou et al., that uses an interval
11 constraint solver to determine the camera position [29, 6, 15]. The constraints are eval-
12 uated in screen-space, where the user defines how the resulting image should appear.
13 The use of interval constraints precludes the use of constraint weighting, which is often
14 useful for controlling camera motion.

15 The constraints are divided into three categories: camera descriptions to control the
16 movement of the camera (panoramic, high/low angle, and fixed location panoramic);
17 image-space constraints on objects (fully included in view, partially included in view,
18 fully excluded from view); and constraints on objects in relation to the camera (orien-
19 tation of the object axis, front side visible, and closer than). Each constraint is assigned
20 a duration, enabling complicated camera paths to be generated by the activation/de-
21 activation of constraints over time. Christie extended the interval constraint approach
22 with the introduction of *constrained hypertubes* for planning the camera path [14, 13].
23 The camera motion is controlled by moving it over pre-defined paths (hypertubes),
24 which are sequenced to generate the camera animation.

25 Halper and Olivier employ the use of genetic algorithms to derive the camera posi-
26 tion, viewing direction, and field-of-view in the CamPlan system [24]. The individuals
27 of the population represent a set of camera properties that are used by the rendering
28 system. The rendered image generated by each individual is used to evaluate the fit-
29 ness (success) of the individual. The scene is rendered in multiple passes, and reads
30 pixel data back from the graphics buffers to determine how well each camera position
31 matches the desired constraints. The execution time of the genetic algorithm is too
32 slow to enable it to be used in real-time interactive applications.

33 Based on [24], Halper et al. explicitly introduced the idea of *frame-coherence* in
34 their camera system developed for computer games [23]. In order to achieve smooth
35 camera movement over time (defined as frame-coherence), path planning approaches
36 are used.

37 Prediction of the future states of the target and camera is used to allow the camera to
38 adapt to future problems [23, 22]. Committing the camera plan to predictive states that
39 may be invalid can produce incorrect camera behaviours. Revising these predictions
40 each frame resolves this issue, however it adds additional computational overheads that
41 are not required by reactive camera systems.

42 To make the camera system suitable for computer games with tight computational
43 requirements, the set of constraints described in [24] is dramatically reduced, leaving
44 only those constraints necessary to maintain control of the camera (relative height,
45 facing, size of object, visibility, and look at).

46 Each constraint is assigned a degree of relaxation, which allows the constraint to be
47 violated before the camera changes its position to satisfy the constraint. As an example,
48 a height value of 100 may have a relaxation value of 10%. If the difference in height
49 between the target and the camera is within $\pm 10\%$ of 100, the camera will not move
50 to satisfy the constraint. Once the difference is greater than 10%, the camera moves to
51 satisfy the constraint. Rather than attempt to progressively satisfy the constraint over
52 time, the camera position is placed at the limit of the relaxation zone of the camera.
53 Because the constraints are evaluated each frame, it is unlikely that the camera moves
54 in inappropriate ways. This approach does not propose a solution for situations where
55
56
57
58

1
2
3
4
5 the camera position is outside the relaxation bounds (over-constrained).

6 Hawkins represents the problem as an optimization problem using stochastic local
7 search [25]. The solver relies on the accuracy of an *initial guess*, which is an estimation
8 of a camera position that satisfies the constraints imposed by the user. The computa-
9 tional time of the solver increases when the accuracy of the initial guess is poor. A local
10 search algorithm using a form of gradient descent is then used to find local optima near
11 the initial guess by minimizing a cost function.

12 Yee and Arabian describe the development of a cinematic camera system suitable
13 for real-time strategy games in [33]. The theory to autonomously select interesting ac-
14 tions and targets based on image-space analysis is presented. Due to the computational
15 expense of these methods, alternative techniques exploiting game logic knowledge to
16 determine the *saliency* of potential objects to visualize is described. Based on this
17 saliency value, potential targets are ordered and then selected for visualization by the
18 cinematic component of the camera system.

19 Camera control methods used by industry are typically very efficient, however they
20 lack expressive power and have limited capacity for adding new capabilities. The exist-
21 ing constraint-based camera control research is primarily focused on cinematography
22 and guided exploration tasks. This renders them unsuitable for use in interactive digital
23 entertainment applications. There is an increasing need for a flexible and extendable
24 camera system that addresses the specific requirements of IDE.
25
26

27 **3 A Weighted CSP Framework**

28
29 The camera system described in this paper is a reactive design, built upon our previous
30 work in [8, 9, 10]. Each frame of animation is treated as a new constraint satisfac-
31 tion problem to be solved, with minimal inter-frame dependencies. This allows the
32 camera to be used in highly dynamic and interactive environments, without additional
33 dependence on the efficiency and robustness of predictive systems.
34

35 The CSP representation used by our framework is simpler than the conventional
36 frameworks described in Section 2.1. We consider only the camera position (X, Y,
37 and Z) as the variables. This eliminates a number of variables from the constraint sat-
38 isfaction process, thereby simplifying the development of efficient constraint solvers.
39 The viewpoint of the camera is typically synchronized with the target position, how-
40 ever extensions to this concept are described in Section 5.2. The manipulation of the
41 field-of-view is not common, so it is removed from the constraint solving process.
42

43 The domain represents a subset of the environment, defining an area in space that
44 the camera can occupy for the current frame. The domain is considered to be cubic,
45 with equal dimensions along each axis. The scale and resolution of the domain is
46 determined by a number of factors, including the distance the camera is expected to
47 move per-frame, and how finely the domain should be searched. The resolution of the
48 search must be sufficient to enable the camera to achieve smooth movement. These
49 parameters are application-specific and are therefore derived accordingly. The domain
50 is centred around the current position of the camera and must be large enough to allow
51 the camera to make the necessary movements for the current frame.
52

53 The constraints used to control the camera are defined and evaluated in world co-
54 ordinates. This allows a straightforward specification of the camera properties, while
55 also simplifying the evaluation process for each constraint. The initial constraint set
56 defines the basic spatial relationship between the camera and the target, and utilizes
57 three constraints (visualized in Figure 1).
58
59
60
61
62
63
64
65

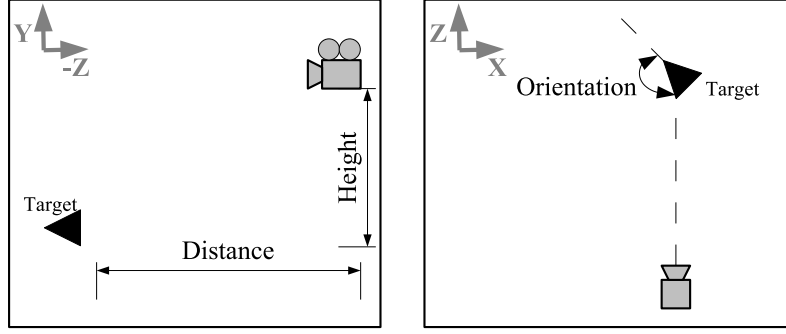


Figure 1: Initial constraint set.

- **Height.** Maintains a relative height relationship with the target.
- **Distance.** Maintains a relative distance relationship with the target.
- **Orientation.** Maintains a relative angular alignment between the camera view vector and the target facing vector.

The boolean (satisfied or unsatisfied) evaluations for each constraint is shown in Equations 1, 2, and 3. The difference in values (Δ values) are derived using Equations 4, 5, and 6

$$height = (height_{desired} - \Delta height) \quad (1)$$

$$distance = (distance_{desired} - \Delta distance) \quad (2)$$

$$orientation = (orientation_{desired} - \Delta orientation) \quad (3)$$

where $height_{desired}$, $distance_{desired}$, and $orientation_{desired}$ are single floating-point values.

$$\Delta height = (camera_y - target_y) \quad (4)$$

$$\Delta distance = \sqrt{((camera_x - target_x)^2 + (camera_z - target_z)^2)} \quad (5)$$

$$\Delta orientation = \left(orientation_{target} \cdot (camera - viewpoint) * \frac{180}{\pi} \right) \quad (6)$$

where $camera$ is the potential camera position being evaluated, and $target$ is the target position. The $target$, $camera$, $viewpoint$, and $orientation_{target}$ variables are all three-component vectors.

3.1 Weighted Constraint Satisfaction

Using boolean constraints forces the camera to maintain the exact spatial relationship as defined by the constraints, producing unrealistic and undesirable camera motion. Rather than applying spring systems to smooth the camera's motion, weighted constraint satisfaction is applied.

Each constraint has a pair of values: a constraint value and a corresponding weight. The constraint value defines the desired spatial relationship between the camera and the target. The constraint weight indicates a relative measure of importance of the constraint. There is no support for hard constraints, and all weights can have arbitrary positive or negative values. Very high weights on a single constraint effectively creates a hard constraint.

The search process is reduced to finding a position in the domain that has the minimal cost in terms of constraint violations. The cost of a constraint is calculated by determining how much the constraint is violated (how unsatisfied it is) and multiplying this value by the constraint weight. The cost of a position (potential solution) is the sum of all constraint costs. To facilitate weighted constraint satisfaction, the initial constraint set must be modified to return a level of satisfaction. The revised evaluations for the constraints are shown in Equations 7, 8 and 9.

$$height_{cost} = |height_{desired} - \Delta height| * height_{weight} \quad (7)$$

$$distance_{cost} = |distance_{desired} - \Delta distance| * distance_{weight} \quad (8)$$

$$orientation_{cost} = |orientation_{desired} - \Delta orientation| * orientation_{weight} \quad (9)$$

where the difference in values (Δ values) is derived using Equations 4, 5, and 6.

Not all constraints return the same scale of violation in their delta values (Equations 4, 5, and 6). Constraints with large delta values contribute more to the overall solution cost when multiplied by their weight. Similar constraint weights should have the same impact on the overall solution regardless of the scale of the delta values. This is achieved by normalizing the constraint weights so that they apply equally to each constraint (Equation 10).

$$NewConstraint_{weight} = OldConstraint_{weight} / Constraint_{value} \quad (10)$$

This equation holds for constraints that can have arbitrarily scaled values (e.g. distance and height). However, the orientation constraint is constrained to degrees, and therefore always has a possible range of 0° through 180° of potential violation. For this reason, when normalizing the orientation constraint, the weight should be divided by 180 rather than $Constraint_{value}$.

Using the weighted constraint satisfaction approach, the camera motion and spatial properties are only weakly coupled. The constraint values define the spatial properties, and do not significantly impact the influence of the constraint weights. The constraint weights effectively define how the camera moves to satisfy the constraints, and do not significantly impact the influence of the constraint values.

The reactive nature of the camera design can make it susceptible to incoherent, or jumpy, movement. Incoherent motion occurs when the distance the camera moves between frames fluctuates wildly, rather than maintaining similar values. This issue is resolved by the introduction of a *frame-coherence* constraint. The constraint evaluates the difference between the distance the camera wants to move in the current frame, and the distance moved in the previous frame. If the difference is large (indicating incoherent motion), the constraint weights the current position to be more expensive than those positions with a smaller difference in the distance metric. This results in a controllable amount coherence in the camera motion.

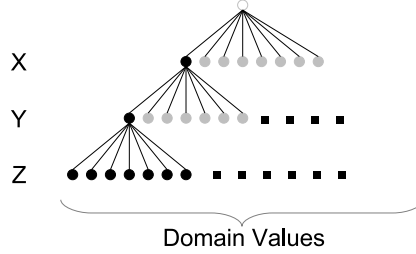


Figure 2: Camera control as a tree search.

Unlike the spatial constraints, the frame-coherence constraint has no preset constraint value. It only has a weight indicating the importance of frame-coherence to the motion of the camera. The frame-coherence evaluation is shown in Equation 11.

$$coherence_{cost} = \frac{|distance_{lastframe} - distance_{potential}|}{frameinterval} * coherence_{weight} \quad (11)$$

where $distance_{potential}$ is the distance the camera will move this frame using the current potential solution. The $frameinterval$ is the elapsed time since the previous frame, allowing the constraint to operate in a frame-rate independent manner. The $distance_{lastframe}$ value is maintained internally by the camera for use in this constraint, and represents the distance the camera moved in the previous frame. The weight of the constraint determines how much variance of movement distance is allowed by the camera.

The task of finding a suitable instantiation for the variables is time-critical in interactive applications. The following section describes the specialized constraint solver developed for camera control in IDE.

4 Constraint Solving

The tree representation of the constraint solving process is shown in Figure 2. The depth of the tree is equivalent to the number of variables used in the representation. The number of nodes for each variable is equal to the number of domain values for each node. As shown in Figure 2, the tree has a very shallow depth (a depth of 3 using our representation). This shallow tree makes the application of backtracking and backjumping heuristics difficult, since the amount of pruning performed by such heuristics is minimal. Getting real-time performance from the solver therefore requires the use of different heuristics.

Our previous work relied on local search methods to improve the computational performance of the constraint solver [8, 9]. By enumerating and examining the search space (or cost surface) for the camera control problem, new algorithms to exploit this information can be devised. Examples of the search spaces are shown in Figure 3.

The cost surfaces in Figure 3 are generated by calculating the cost of each solution within the searchable domain, evaluating only the height, orientation, and distance constraints with a static target, facing along and positioned along the positive Z axis. The box represents the searchable domain for the camera position, with the colour of each potential solution representing the cost. Solutions with a low cost (preferable) are

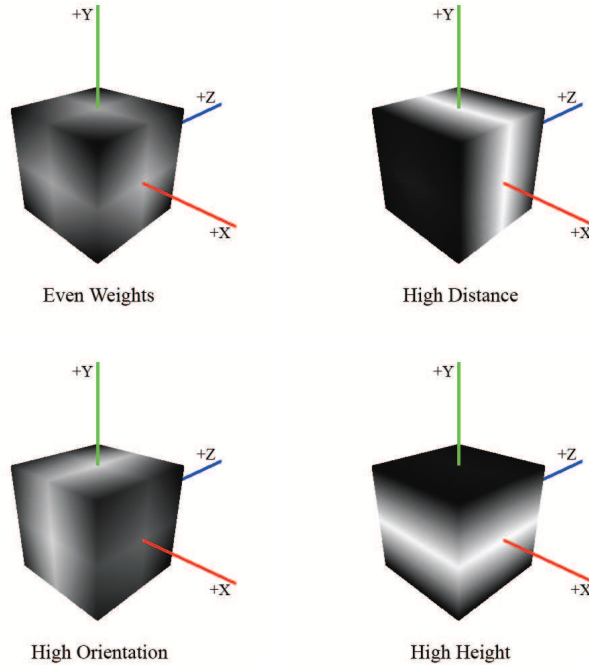


Figure 3: Example cost surfaces for the weighted constraint satisfaction representation.

shown in white, while solutions with a high cost (undesirable) are shown in black. The strength of the gradient illustrates how the constraint weighting influences the search space.

High constraint weights result in much sharper gradients, providing a clear delineation between unacceptable constraint violations and constraint satisfaction. Low constraint weights result in broader gradients, providing a gradual transition between unsatisfactory and satisfactory values. With broader gradients, the camera makes gradual movements to satisfy the constraint(s), resulting in a *relaxed* camera motion, whereas sharper gradients cause the camera to make fast movements to satisfy the constraint(s), resulting in an *energetic* camera motion.

In the top-left graph in Figure 3, all of the constraints have an equal weight. The remaining graphs illustrate the influence of applying a higher weight to a single constraint. Although the optimal position appears to be in the centre of each search space, this is primarily due to the manner in which the figures were generated. The movement of the target and implementation of different constraints causes the optimal solution to be located in various regions of the search space. Therefore, simply starting the search from the mid-point of the search space is no guarantee of finding the optimal solution quickly.

4.1 Exploiting Problem Structure

Based on the information obtained about the search space in the previous section, a customized constraint solver was devised. An early version of this solver is described in [11]. The solver, *Sliding Octree*, is based on the octree spatial data structure shown

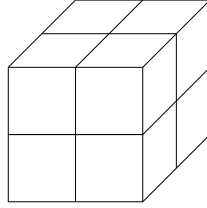


Figure 4: The octree spatial data structure.

```

15 calculate camera facing and right vectors;
16 calculate initial domain size;
17 set centre of octree to current camera position;
18 while current pass less than maximum passes do
19     | configure octree nodes;
20     | for each octree node do
21     |     | evaluate octree node as potential solution;
22     |     | if octree node cost less than best solution cost then
23     |     |     | keep octree node as best solution;
24     |     | end
25     | end
26     | reduce domain size by scaling factor;
27     | slide direction = (best solution - octree centre);
28     | new octree centre += slide direction * domain size;
29     | increment pass count;
30 end
31 return best solution as new camera position;

```

Algorithm 1: The Sliding Octree solver.

in Figure 4.

In the sliding octree solver, the octree encapsulates the entire searchable domain. This requires the domain to be cubic, with each axis having equal dimensions. The centre point of each of the eight nodes of the octree is evaluated as a potential solution. These nodes provide cost information for spatially diverse and consistent regions of the search space.

Before the simulation starts, the initial camera position (frame 0) needs to be calculated. This is done by explicitly solving all of the constraints to place the camera at the most appropriate position. For example, the height of the camera is explicitly set the desired value of the height constraint, with this process repeated for all of the constraints. Once this initial position has been calculated, the constraint solver can be executed to derive the optimal position according to the full constraint/weighting set.

The pseudo-code for the sliding octree solver is shown in Algorithm 1. The search starts by generating an octree that encapsulates the entire domain. The dimensions of the domain (bounding box) for the search space is equivalent to the maximum distance the camera can move in a single frame. The mid-point of each node is then evaluated as a potential camera position. If any of the eight nodes are better (lower overall cost) than our best known solution, we update the best known solution. At the end of each pass, the octree is scaled down and slid towards the best known solution, to refine the search around the optimal solution. This process is repeated for a preset number of

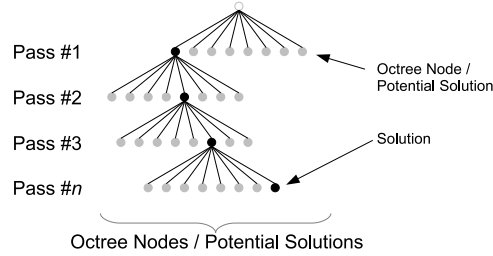


Figure 5: The search tree for the sliding octree solver.

passes until the final camera position is determined. The new camera position is then tested against the environment for collision as a post-processing step (collision is not part of the constraint solving process).

The resulting search behaviour starts coarsely, obtaining cost information about sparse regions of the search space. As the search progresses, scaling the octree down forces a more focused sampling of possible solutions. By sliding the octree towards the best known solution, the focused portion of the search occurs in the region most likely to contain the optimal solution. The octree slides towards the best known solution on each pass, allowing the search to move and adapt to the increased information about the optimal solutions as the search progresses. This results in the solver effectively pruning large portions of the search space at the start of the search, while providing a high resolution search in the region containing the optimal solutions at the end of the search.

The scaling factor of the octree and the number of passes are the parameters to the solver. If the scaling factor is too large, the octree shrinks too quickly and large portions of the search space may be missed. If the scaling factor is too small, the solver takes significantly longer to converge on the optimal solution. A generally useful scaling factor is to reduce the size of the octree by 25% on each pass.

If the number of passes is too small, the solver stops before converging on the optimal solution. Although the number of required passes is application specific, a general rule is to have three times the domain size for the number of passes (Equation 12).

$$passes = size_{domain} * 3 \quad (12)$$

The sliding octree solver is in essence a best-first search, following the node with the lowest solution cost at the end of each pass. There is always a fixed number of evaluations on each pass (eight), and the search continues to follow the best path towards the solution.

In some instances it is beneficial to enumerate multiple solutions for a given frame. This is typically used in cinematography systems, where a higher-level decision system is responsible for selecting the best camera position from a set of candidate positions. The search tree generated by the sliding octree solver is shown in Figure 5. The revised search tree is more balanced than the original tree representation (Figure 2), with the depth of the tree now equal to the number of passes of the solver. The increased depth of the tree enables the application of backtracking heuristics to improve performance when enumerating multiple solutions.

The pseudo-code for the backtracking sliding octree solver is shown in Algorithm 2. The major difference in this variation of the algorithm is the maintenance of a list

```

1 initialize and clear the ordered solution list;
2 calculate camera facing and right vectors;
3 calculate initial domain size;
4 set centre of octree to current camera position;
5 while current pass less than maximum passes do
6     configure octree nodes;
7     evaluate octree nodes as potential solutions;
8     for each octree node do
9         insert pass number, node position, and node cost into ordered solution
10        list;
11        if octree node cost less than best solution cost then
12            | keep octree node as best solution;
13        end
14    end
15    if all octree nodes cost more than best solution then
16        | pop the next best solution from the ordered solution list;
17        | backtrack the search to restart using next best solution;
18    end
19    reduce domain size by scaling factor;
20    slide direction = (best solution - octree centre);
21    new octree centre += slide direction * domain size;
22    increment pass count;
23 end
24 return best solution as new camera position;

```

Algorithm 2: The Backtracking Sliding Octree solver.

of solutions that are ordered by cost, with new optimal solutions added to the front of the list. These solutions are used as restart points when the algorithm backtracks (or backjumps), and it is only necessary to store as many as necessary for the solver to run. The determination of how far and when to backtrack has not yet been fully examined, however the search termination condition must be clearly defined to prevent the search turning into a systematic complete method.

4.2 Results

To evaluate the effectiveness of the sliding octree solver, a comparison study was conducted using three different constraint solver methodologies over the same animation. Each of the graphs in Figure 6 represents the movement distance made by the camera each frame. The distance-per-frame metric provides a simple measure of movement coherence without analysing the 3D movement graph of the camera through the environment. Large changes in movement distance between subsequent frames indicate erratic and incoherent camera motion.

Figure 10 shows some sample screenshots of the animation, which consisted of a single target moving through a dynamic environment. The constraints applied were height, distance, orientation, frame-coherence and occlusion (described in Section 5.1).

The per-frame distance the camera moves through the environment over a given sample animation is used as the testing criteria. Each solver is run over the same animation using the same constraint parameters. All experiments were conducted on an Athlon 2800+ with 1Gb of RAM, running Windows XP Service Pack 2. The distance

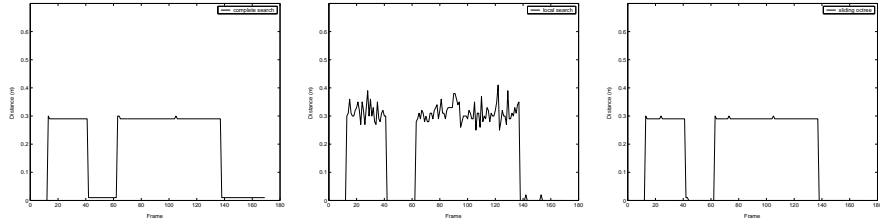


Figure 6: Visual results from each constraint solver.

Property	Complete Search	Local Search	Sliding Octree
Domain Size (per-axis)	± 1.0	± 1.0	± 1.0
Resolution	0.0005	N/A	N/A
Solutions Evaluated (per frame)	64,000,000,000	3,000	240
Average Time per Solution (secs)	2,655.50	0.00025	0.00002
Correlation Value (r)	1.0	0.987176	0.999784
Passes	N/A	N/A	30
Scale Factor	N/A	N/A	0.75

Table 2: Configuration and results of the revised solver tests.

the camera moves each frame is recorded, with the results shown in Figure 6.

The graphs in Figure 6 demonstrate the distance the camera moved per-frame over the sample animation. The left graph shows the movement properties generated by the generate-and-test solver, and represents the optimal graph for camera motion. The middle graph shows the movement of the local search algorithm. The amount of jumpiness is due to the randomized nature of the search. This solver is capable of producing reasonable visual quality with real-time computational performance.

The right graph shows the movement generated by the sliding octree solver. The shape of the graph closely matches that of the generate-and-test method. This demonstrates that the sliding octree solver is able to generate comparable visual quality to the generate-and-test method, while achieving faster performance than the local search algorithm. The configuration and results of the solvers is shown in Table 2. The computational time demonstrates the real-time efficiency of the sliding octree solver in comparison to the other methods.

The correlation value represents how closely the movement graphs match. Correlation values of 1.0 represent a perfect correlation (same graph), whereas correlation values of 0.0 represent no correlation. The correlation value confirms that the sliding octree solver generates comparable visual results (0.999784) to the complete method.

Differences in the constraint representation, implementations, environments, and capabilities make it impossible to fairly compare our camera system against existing works. The only comparable constraint solver that exploits the spatial nature of the problem is the recursive grid solver by Bares et al. [2]. Our sliding octree solver is able to find the solution within 240 evaluations (8 nodes per-pass, 30 passes), while the Bares et al. recursive grid solver requires 729 evaluations on the first pass (grid size of 9x9x9). The recursive grid solver considers more parameters (such as field-of-

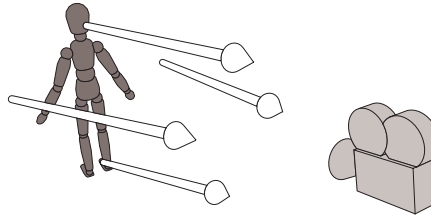


Figure 7: The four rays that are cast through the environment to extract occlusion information.

view) and potentially larger regions of space, whereas our constraint representation is simplified for application in IDE.

5 Extended Functionality with Constraints

The previous sections described a weighted CSP framework for camera control, along with some simple spatial constraints. The capabilities of the camera system can be easily extended by introducing new constraints. The elegance of the CSP framework allows these new capabilities (constraints) to be added without modifying the optimized constraint solver. The only requirement on new constraints is that they return a degree of satisfaction for use in the weighted CSP framework.

To demonstrate how these extensions are applied, they are described in relation to a sample game. The purpose of this game is to find lost treasure in the valley, and involves the player negotiating hazards that are specifically designed to stress-test the camera system. Additional characters are available for interaction, and the player can also eavesdrop on three characters discussing their plot to start a war (demonstrating cinematics).

5.1 Occlusion Avoidance

One of the most critical abilities of an autonomous camera is maintaining a clear line-of-sight between the camera and its target. Maintaining this line-of-sight is a difficult problem, due to the dynamic and interactive nature of virtual environments for IDE. The camera system must react quickly to dynamic objects that occlude the users view. This is achieved by introducing a new *occlusion* constraint.

In order to avoid potential occlusions, the location of occluders in the environment must be efficiently determined. Figure 7 shows the four rays that are cast into the environment from the target to determine occlusion on each frame. Their arrangement (top, bottom, left, right) is designed to obtain information about the environment immediately surrounding the camera and target, although any arrangement or number of rays can be used. If any of the rays intersect with a part of the environment (world geometry), then the camera has a risk of occlusion.

This method does not build an explicit detailed occlusion map, instead providing indications of potential occluders that may impede the line-of-sight to the target. Dense environments filled with small occluders may require the use of more rays to build determine the potential for occlusion. Using this potential occlusion information, adjustments to the camera position can be instigated before the occluder blocks the line-of-sight to the target.

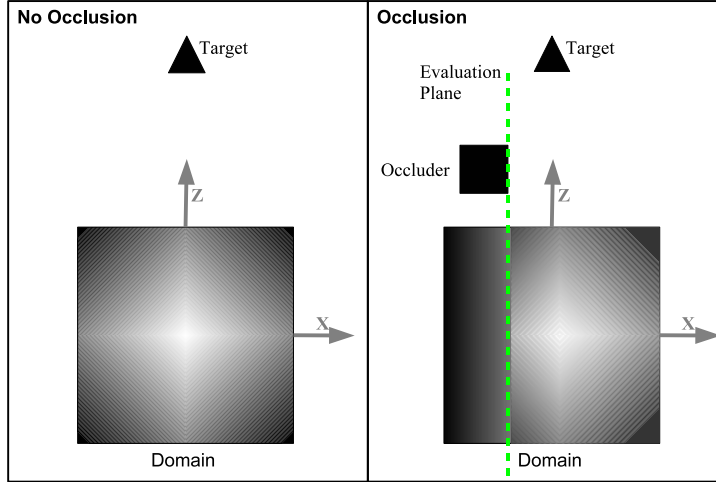


Figure 8: The manipulation of the cost surface by the occlusion constraint.

When a ray is flagged as occluded, the occlusion constraint makes potential positions close to the occluded ray more expensive in terms of solution cost. By making these positions more expensive, the constraint solver is less likely to select them as the camera position for the current frame, thereby avoiding occlusions. Figure 8 shows an example of how the solution cost is manipulated to avoid potential occlusions. The left figure shows a typical cost surface (viewed from above) with no occlusion, where the optimal position is in the centre of the search space.

The right figure demonstrates the addition of an occluder that would be detected by an intersection with the left ray (Figure 8). The resulting cost surface shows that solutions that are closer to the occluder, and therefore in danger of being occluded, are more expensive in terms of solution cost. The occlusion constraint evaluation is shown in Equation 13.

$$occlusion_{cost} = distance(camera_{position}, ray_{intersection}) * occlusion_{weight} \quad (13)$$

where $camera_{position}$ is the potential camera position, and $ray_{intersection}$ is the world coordinates position of the ray intersection with the environment. The function $distance$ returns the planar distance.

The necessary use of planar distance calculations is shown in Figure 9. In the left figure, a top-down view of the Euclidean distance calculation is shown. In this case, even though the potential position lies in an occluded position, the distance calculation fails to weight the position correctly. This can lead to situations where the camera position is occluded, despite the rays flagging potential occlusions. The equivalent planar distance calculation is shown in the right figure. Solutions that are in-line with the occluded position (determined by the evaluation plane), are equally distant from the plane regardless of their Z axis values. This generates the expected camera behaviour.

As the occlusion constraint is weighted, it works in conjunction with the other constraints evaluated by the camera system. This allows the occlusion constraint to be violated as necessary by the spatial constraints. The weights applied to each constraint determine the characteristic behaviour of the camera when making a trade-off between conflicting constraints.

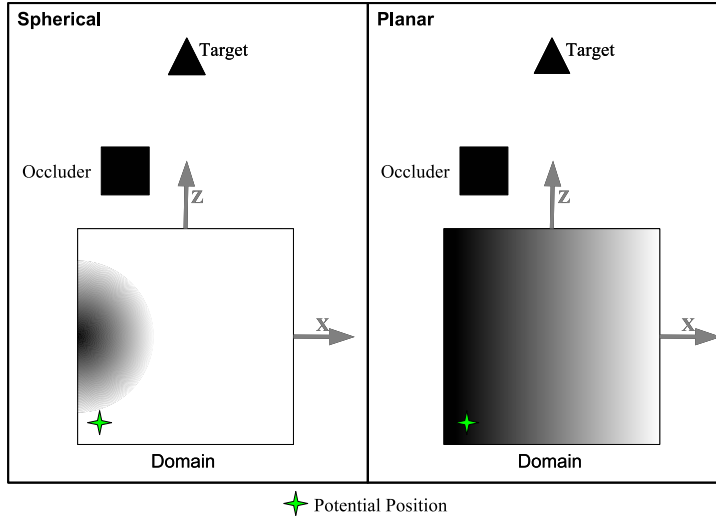


Figure 9: Spherical versus Planar distance calculations.

Therefore, the movement of the camera to avoid occlusion can be controlled by manipulating the weight of the occlusion constraint in relation to the weights of the spatial constraints. In some situations, the occlusion constraint can cause *thrashing*. This occurs when the intersection between a ray and the environment toggles between consecutive frames. To alleviate this problem, the frame-coherence constraint can be assigned a higher weight, eliminating the small thrashing movements.

5.1.1 Occlusion Scenario #1

In this scenario, the character is forced to walk through a low tunnel (Figure 10 top-left). Due to the height of the camera, its view will become occluded unless it adjusts its height. The top ray becomes occluded, causing the camera to reduce its height until the occlusion risk (ray intersection) is removed. At this point, the camera maintains an appropriate height (no occlusion) until the character is free from the obstacle (Figure 10 top-right). This is a simple and very common scenario in third-person games, however as environment complexity increases, so does the risk of occlusion.

5.1.2 Occlusion Scenario #2

In this scenario, the character is moving through a very cramped room, with very little room for the camera to move up/down, left or right. There is really no suitable camera position to resolve the desired constraints, so the use of a *safety point* is employed. This point provides a catch-all camera position that is unlikely to encounter occlusion. As such, it is usually positioned quite close to the character, to reduce the chances of occlusion. When all of the rays are occluded (Figure 10 bottom-right), the occlusion constraint attempts to move to the safety point to resolve the occlusion. The cost of the solution is reduced the closer the camera is to this point. Once some (or all) of the occlusion problems are resolved, the camera returns to satisfying its constraints (Figure 10 bottom-left).

The use of the safety point allows the camera to handle complex and cramped



Figure 10: Demonstration of occlusion avoidance behaviour.

environments, while providing suitable camera positions. This allows the simple occlusion evaluation technique to be employed, reducing system complexity (and testing/debugging time). An example is the requirement that the camera will attempt to follow the character through a window. While doing so, it is likely that the occlusion rays will all become occluded, luring the camera to the safety point. This effectively *draws* the camera through the window, maintaining visibility of the character at all times.

5.2 Multiple-Target Visualization

In interactive digital entertainment, the main character must interact with environmental objects, non-player characters (NPCs), or other external entities. It is often important that the user is able to visualize these objects as well as their character in order to maintain the spatial relationships between them. By having the camera subtly move the viewpoint in the general direction of the other objects of interest, the camera can assist the user in their goals.

When visualizing multiple targets, an alternative viewpoint calculation method is to use a weighted average calculation. A new viewpoint position is calculated based on a weighted average between the primary target and any additional targets in the environment. Each target is assigned weight indicating its preference to be in the centre of the resulting camera view. The player's avatar is assigned a higher weight to indicate that it should be the focus.

The viewpoint position is recalculated each frame, so additional targets can be dynamic, or added and removed on a per-frame basis. The viewpoint is interpolated using an ease-in/ease-out method [30], allowing the viewpoint to smoothly transition

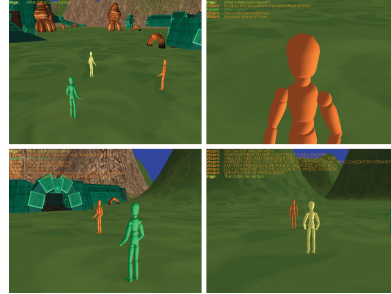
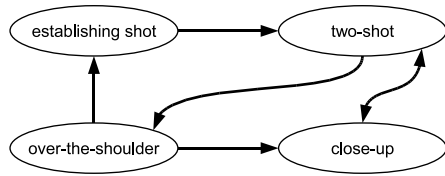


Figure 11: Example state transitions between camera profiles and the resulting visualisation.

between the current and desired positions.

Determining which targets the user is interested in viewing is a large challenge, which is often domain specific (Yee and Arabian provide some guidelines in [33]). A simple method that works well in a general case is to add any target to the target list using the *half-distance rule*. This rule allows any object to be evaluated by the camera if it enters within a proximity (radius) of the primary target. The radius of the proximity is half the value of the distance constraint desired value.

The weighted average viewpoint method works in conjunction with camera profiles to provide the camera with cinematic capabilities.

5.3 Cinematography and Replays

A camera profile defines the configuration for a given *state* of the camera, similar to a camera module in [18, 20, 19]. The camera profile includes the constraint values and weights, and also a list of which constraints are to be evaluated by the constraint solver. Suitable camera profiles can be derived automatically to reproduce an example animation [10]. The camera profile is the interface between the camera system and the artist/designer who is tasked with defining the camera behaviours. This abstracts a lot of the technical and low-level configuration of the constraint solver from the interface.

Camera profiles can be switched dynamically during the simulation, allowing constraints to be toggled on/off over time. The constraint values and weights can be changed between evaluation frames without detriment to the camera motion. The constraint solver implicitly interpolates the camera position, moving according to the weights as specified in the newly instantiated profile. This removes the need for explicit interpolation of camera properties for transition effects. The frame-coherence constraint ensures that the camera motion is smooth during the interpolation to the new profile.

The location of the camera viewpoint needs to be calculated differently for cinematics. It is calculated as a weighted average between the positions of the actors involved in the scene, with the currently speaking actor assigned a higher weight. For monologues with a single actor, the viewpoint location is synchronised to the actor position. The line-of-action (the line between two interacting actors) is used as the *facing* vector for alignment with the orientation constraint.

To simplify the specification of cinematic or replay sequences, a finite-state-machine based method of camera profile switching is used. This approach is similar to the module-based approach in [19] and the hierarchical finite-state-machine approach in

1
2
3
4
5 [32]. In our approach the camera states are not hierarchically organized (left side of
6 Figure 11).

7 Each state represents a camera profile, whose properties dictate the replication of a
8 desired shot type. Cinematic (or replay) rules are encoded for the entry/exit conditions
9 of each state. This allows cinematography rules and conventions to be naturally and
10 concisely specified. Each state uses a set of rules to determine possible state transitions
11 and shot durations. For event-based transitioning, scripting systems that respond to
12 in-game triggers can be used to force a state transition in relation to events occurring
13 in the environment.

14 State transitions immediately change the active camera profile being used by the
15 constraint solver. The natural interpolation performed by the solver can be used as
16 the visual transition, or immediate camera state changes (a cut) can be used. The
17 camera performs coherent movement and effectively avoids occlusions as defined by
18 the camera profile associated with the current state.

19 Although the basic constraint set (height, distance, orientation, frame-coherence,
20 and occlusion), combined with the concept of profiles provides some flexibility in
21 viewing definitions, it is not sufficient for all cinematic requirements. In these
22 cases, it is necessary to derive new constraints that specifically fulfill these require-
23 ments. Hawkins provides a number of suitable constraints that can be adapted for use
24 in our camera framework in [25].
25
26

27 **5.3.1 Cinematic Scenario**

28 To demonstrate the cinematic capabilities of our approach, the player can listen in to a
29 scene in which three characters are discussing how to start a war. Each character has
30 dialog, with some characters delivering extended monologues. The state machine used
31 to visualise this scene is shown in the left side of Figure 11.
32

33 Screenshots from the scene are shown in the right side of Figure 11. The top-left
34 figure demonstrate the establishing shot used at the start of the scene, enabling the
35 viewer to spatially recognise each characters position. The top-right figure shows a
36 close-up view of a character delivering a monologue. The bottom figures demonstrate
37 a two-shot, where one character is directly addressing another.

38 The transitions of the camera from speakers to listeners, and between the different
39 shots is believable. A more detailed state-machine would provide for more variance in
40 the way the scene is visualized. The transitions between cinematic shots is done using
41 camera cuts, with some panning during shots provided automatically by the camera as
42 the conversation continues. All of the shots are configured using the base constraints.
43
44

45 **6 Discussion**

46
47 By visualizing and analyzing the search space, the development of a specialized and
48 efficient solver technique for camera control was possible. The sliding octree solver
49 shares similar characteristics to the recursive grid solver [2] and the semantic space
50 partitioning approach of [16], by exploiting the spatial nature of the problem. In the
51 recursive grid solver, the best candidate position per-pass is used as the centre point
52 of a refined search. Many candidate positions must be evaluated in the early passes to
53 ensure the higher resolution search (with fewer candidate positions) is performed in the
54 region containing the optimal solution. With the sliding octree solver, the octree slides
55
56
57
58
59
60
61
62
63
64
65

1
2
3
4
5 towards the best found solution (since the search started, not per-pass) over a larger
6 number of passes with fewer evaluations per-pass.

7 By conducting the search over more passes with smaller movements through the
8 search space, the sliding octree algorithm is less susceptible to missing optimal solu-
9 tions that don't exist within the immediate vicinity of the sample points in the early
10 passes. In order for the recursive grid solver (in the form described in [2]) to attain
11 the same robustness, many more sample points per-pass are required, which increases
12 the number of positions evaluated throughout the search. Using an example from [2],
13 using a 9x9x9 grid on a first pass requires a total of 729 evaluations before moving to
14 the refinement passes. This is in comparison to the total of 240 evaluations required by
15 the sliding octree solver to find the final camera position (Table 2).

16 Constraints can be added to expand the capabilities of the camera, as demonstrated
17 by the introduction of the occlusion constraint. The addition of new constraints to the
18 camera system does not require the constraint solver to be modified, so the efficient
19 sliding octree solver can be applied regardless of the constraints in use. The only
20 requirement on the introduction of new constraints is that they return a measure of
21 satisfaction.

22 Camera control systems designed specifically for IDE [23] rely upon predictive
23 systems to maintain coherent camera motion over time. The reliance on predictive
24 techniques has the drawback of the cost of prediction (in both computational time and
25 implementation complexity), and also restricts the reliability of the camera system to
26 that of the predictive system. The reactive design of our camera system is preferable
27 due to its flexibility, however initial tests demonstrated it could be prone to incoherent
28 motion. The use of the frame-coherence constraint eliminates this problem.

29 The reactive design allows the camera to be used in highly dynamic and interactive
30 environments. In comparison to predictive camera systems, with our reactive design
31 the environment can be totally changed between evaluation frames without detriment
32 to the camera's operation.

33 The weighted constraint satisfaction framework demonstrated the decoupling of
34 the spatial camera properties (constraint values), and the camera movement properties
35 (constraint weights). In contrast to the weighted framework described by Bares et al
36 [5], our representation does not have the concept of *hard* constraints. Each constraint
37 is susceptible to violation as necessary, however setting a sufficiently high weight can
38 effectively create a hard constraint. This feature enables the camera to recover from
39 over-constrained situations without the need for a specialized recovery strategy. By
40 carefully manipulating the constraint weights, excellent control over the camera motion
41 can be accomplished.

42 The set of constraint values and weights are grouped into camera profiles, which are
43 similar to the camera modules described by Drucker [18, 20, 19]. Each camera profile
44 defines the camera properties for a given period of time. These profiles can be switched
45 arbitrarily at run-time, with the constraint solver interpolating between the old and new
46 camera profiles automatically. Camera profiles provide a convenient correlation to
47 states in cinematography, making the specification of cinematatics a process of defining
48 sequences of camera profiles.

49 The application of cinematography to IDE was not the primary focus of this re-
50 search. Although the finite-state-machine cinematic system described in Section 5.3
51 is usable, there are more effective cinematography solutions (discussed in Section 2).
52 The techniques described in this paper deal with the low-level problem of finding suit-
53 able camera positions to satisfy viewing criteria through constraints. For cinematic
54 purposes, a higher-level framework is often preferred. We view our work as compli-
55
56
57

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

mentary to higher-level automated cinematography research, by allowing such systems to be built upon a flexible and reliable constraint-based camera system.

The camera system described in this paper achieved the requirements of an autonomous camera system for IDE (Section 1.1):

- **Real-time.** Our results (Table 2) demonstrate an average evaluation time of 0.0002 seconds/frame.
- **Competent.** The combination of all of the constraints demonstrate the competence of the camera, as described in the scenarios in Section 5.1.
- **Extendable.** The nature of constraint-based approaches allows extendability through the introduction of new constraints.
- **Dynamic.** The reactive design of our camera system allows the environment and targets to be fully dynamic, without unnecessary pre-planning or inter-frame dependencies.
- **Environment independence.** All environmental information (occlusion data) is determined reactively by the camera system per-frame, and can operate in arbitrary environments with no prior knowledge of the environment structure.
- **Ease of implementation.** The camera does not use any pre-determined information about the environment, and obtains relevant information reactively at runtime.
- **Cinematic.** The combined use of constraints and a finite-state-machine approach allow cinematics to be accomplished, as described in the scenario in Section 5.3.

7 Conclusion

This paper has described a novel application of constraint satisfaction to address the outstanding and difficult practical problem of autonomous camera control for interactive digital entertainment. The state-of-the-art in constraint-based camera systems was surveyed, and provided a foundation for the design of a reactive, weighted constraint-satisfaction based camera system. An analysis of the search space for the problem domain led to the development of an optimized constraint solver for this particular representation. The flexibility of the camera system design was illustrated by adding new capabilities through additional constraints.

Extensions to the camera system that provide suitable control for complex cinematography and replay requirements was described. The extensions were combined with a state-based architecture for specifying cinematic knowledge to be applied in real-time.

7.1 Future Research

There are several directions for future research based on this work. An empirical study to determine the optimal parameters for scaling and sliding of the sliding octree solver can prove valuable. The derivation of specialized heuristics for the backjumping variant of the solver can provide significant performance improvements when enumerating multiple solutions. Dynamically determining the mood of the scene for use by the cinematography system, and applying this knowledge to transition between suitable

1
2
3
4
5 camera profiles remains an open research problem. Switching camera profiles can lead
6 to choppy transitions, so further refinements to profiles transitioning would improve
7 the cinematic capabilities of the camera.

8 The differences between camera system representations makes it difficult to effectively
9 evaluate new camera control methodologies. The creation of a standard set of
10 problems, visualization capabilities, and standard implementations would benefit the
11 research area significantly. A consistent set of criteria used to measure improvements
12 in camera control techniques will make advances in this area easier to judge. Detailed
13 user testing of the camera system in different IDE environments will provide conclusive
14 proof of the effectiveness of our solution.

15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65

- [1] Daniel Arijon. *Grammar of the Film Language*. Silman-James Press, September 1991 (originally published 1976).
- [2] William Bares, Scott McDermott, Christina Boudreaux, and Somying Thainimit. Virtual 3D Camera Composition from Frame Constraints. In *Proceedings of 8th ACM International Conference on Multimedia*, pages 177–186, Los Angeles, CA, USA, October 30 - November 3 2000.
- [3] William H. Bares, Joël P. Grégoire, and James C. Lester. Realtime Constraint-Based Cinematography for Complex Interactive 3D Worlds. In *Proceedings of Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI)*, pages 1101–1106, Madison, WI, USA, July 26-30 1998. AAAI Press.
- [4] William H. Bares and James C. Lester. Intelligent Multi-Shot Visualization Interfaces for Dynamic 3D Worlds. In *Proceedings of 1999 International Conference on Intelligent User Interfaces*, pages 119–126, Los Angeles, CA, USA, January 5-8 1999.
- [5] William H. Bares, Somying Thainimit, and Scott McDermott. A Model for Constraint-Based Camera Planning. In *Proceedings of AAAI Spring Symposium on Smart Graphics*, pages 84–91, Stanford, CA, USA, March 20-22 2000.
- [6] Frédéric Benhamou, Frédéric Goualard, Éric Languénou, and Marc Christie. Interval Constraint Solving for Camera Control and Motion Planning. *ACM Transactions on Computational Logic*, 5(4):732–767, October 2004.
- [7] Nick Bobick. Rotating Objects Using Quaternions. <http://www.gamasutra.com/features/19980703/quaternions.01.htm>, July 3 1998.
- [8] Owen Bourne and Abdul Sattar. Applying Constraint Satisfaction Techniques to 3D Camera Control. In Geoffrey I. Webb and Xinghuo Yu, editors, *Proceedings of 17th Australian Joint Conference on Artificial Intelligence*, pages 658–669, Cairns, Australia, December 4-6 2004. Springer.
- [9] Owen Bourne and Abdul Sattar. Applying Constraint Weighting to Autonomous Camera Control. In *Proceedings of The First Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 3–8, Marina Del Ray, CA, USA, June 1-3 2005. AAAI Press.

- 1
2
3
4
5 [10] Owen Bourne and Abdul Sattar. Evolving Behaviours for a Real-Time Au-
6 tonomous Camera. In Yusuf Pisan, editor, *Proceedings of 2nd Australasian Con-*
7 *ference on Interactive Entertainment*, pages 27–33, Sydney, Australia, November
8 23-25 2005.
- 9
10 [11] Owen Bourne and Abdul Sattar. Autonomous Camera Control with Constraint
11 Satisfaction Methods. In Steve Rabin, editor, *AI Game Programming Wisdom 3*,
12 pages 173–187. Charles River Media, March 2006.
- 13
14 [12] Phil Carlisle. An AI Approach to Creating an Intelligent Camera System. In
15 Steve Rabin, editor, *AI Game Programming Wisdom 2*, pages 179–185. Charles
16 River Media, December 1 2003.
- 17 [13] Marc Christie and Éric Languéno. A Constraint-based Approach to Camera
18 Path Planning. In Andreas Butz, Antonio Krüger, and Patrick Olivier, editors,
19 *Proceedings of 3rd International Symposium on Smart Graphics*, pages 172–181,
20 Heidelberg, Germany, July 2-4 2003. Springer.
- 21
22 [14] Marc Christie, Éric Languéno, and Laurent Granvilliers. Modeling Camera Con-
23 trol with Constrained Hypertubes. In Pascal Van Hentenryck, editor, *Proceedings*
24 *of 8th International Conference on Principles and Practice on Constraint Pro-*
25 *gramming*, pages 618–632, Ithaca, NY, USA, September 9-13 2002.
- 26
27 [15] Marc Christie, Rumesh Machap, Jean-Marie Normand, Patrick Olivier, and
28 Jonathan Pickering. Virtual Camera Planning: A Survey. In *Proceedings of*
29 *5th International Symposium on SmartGraphics*, Frauenwoerth, Germany, Au-
30 gust 22-24 2005. Springer.
- 31
32 [16] Marc Christie and Jean-Marie Normand. A Semantic Space Partitioning Ap-
33 proach to Virtual Camera Control. In *Annual Eurographics Conference, Com-*
34 *puter Graphics Forum*, pages 247–256, Dublin, Ireland, August 29-September 2
35 2005.
- 36
37 [17] Andrew Corrigan. A Simple Third-Person Camera Using The Polar Coordinate
38 System. <http://www.gamedev.net/reference/articles/article1591.asp>, November
39 13 2001.
- 40
41 [18] Steven M. Drucker and David Zeltzer. Intelligent Camera Control in a Virtual
42 Environment. In *Proceedings of Graphics Interface '94*, pages 190–199, Banff,
43 Alberta, Canada, May 18-20 1994.
- 44
45 [19] Steven M. Drucker and David Zeltzer. CamDroid: A System for Implementing
46 Intelligent Camera Control. In *Proceedings of 1995 Symposium on Interactive*
47 *3D Graphics*, pages 139–144, Monterey, CA, USA, April 9-12 1995.
- 48
49 [20] Steven Mark Drucker. *Intelligent Camera Control for Graphical Environments*.
50 PhD thesis, Massachusetts Institute of Technology, June 1994.
- 51
52 [21] John Giors. The Full Spectrum Warrior Camera System. In *Proceedings of Game*
53 *Developers Conference 2004*, San Jose, CA, USA, March 22-26 2004.
- 54
55 [22] Nicolas Halper. *Supportive Presentation for Computer Games*. PhD thesis, Uni-
56 versity of Magdeburg, October 17 2003.
- 57
58
59
60
61
62
63
64
65

- 1
2
3
4
5 [23] Nicolas Halper, Ralf Helbing, and Thomas Strothotte. A Camera Engine for
6 Computer Games: Managing the Trade-Off Between Constraint Satisfaction and
7 Frame Coherence. *Computer Graphics Forum*, 20(3):174–183, September 2001.
8
9 [24] Nicolas Halper and Patrick Olivier. CAMPLAN: A Camera Planning Agent.
10 In *Proceedings of AAAI Spring Symposium on Smart Graphics*, pages 92–100,
11 Stanford, CA, USA, March 20-22 2000.
12
13 [25] Brian Hawkins. *Real-Time Cinematography for Games*. Charles River Media,
14 January 28 2005.
15
16 [26] Dante Treglia II. Camera Control Techniques. In Mark DeLoura, editor, *Game*
17 *Programming Gems*, pages 371–379. Charles River Media, August 2000.
18
19 [27] Vipin Kumar. Algorithms for Constraint Satisfaction Problems: A Survey. *Artifi-*
20 *cial Intelligence Magazine*, 13(1):32–44, 1992.
21
22 [28] John E. Laird and Michael van Lent. Human-Level AI’s Killer Application: In-
23 teractive Computer Games. In *Proceedings of Seventeenth National Conference*
24 *on Artificial Intelligence and Twelfth Conference on Innovative Applications of*
25 *Artificial Intelligence*, pages 1171–1178, Austin, TX, USA, July 30 - August 3
26 2000. AAAI Press.
27
28 [29] E. Languéno, F. Benhamou, F. Goualard, and M. Christie. The Virtual Camera-
29 man: an Interval Constraint Based Approach. In *Proceedings of 13th European*
30 *Conference on Artificial Intelligence (Constraint Techniques for Artistic Applica-*
31 *tions Workshop)*, Brighton, United Kingdom, August 23-28 1998.
32
33 [30] John Olsen. Interpolation Methods. In Mark DeLoura, editor, *Game Program-*
34 *ming Gems*, pages 141–149. Charles River Media, August 2000.
35
36 [31] Jonathan Stone. Third-Person Camera Navigation. In Andrew Kirmse, editor,
37 *Game Programming Gems 4*, pages 303–314. Charles River Media, March 1
38 2004.
39
40 [32] Li wei He, Michael F. Cohen, and David H. Salesin. The Virtual Cinematogra-
41 pher: A Paradigm for Automatic Real-Time Camera Control and Directing. In
42 *Proceedings of 23rd Annual Conference on Computer Graphics (SIGGRAPH 96)*,
43 pages 217–224, New Orleans, LA, USA, August 4-9 1996. ACM Press.
44
45 [33] Hector Yee and Elie Arabian. Battle Cam: A Dynamic Camera System for Real-
46 Time Strategy Games. In *Game Developers Conference*, San Jose, CA, USA,
47 March 24 2006.
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65