HAL
open science

Accurately Measuring the Satisfaction of Visual Properties in Virtual Camera Control

Roberto Ranon, Marc Christie, Tommaso Urli

To cite this version:

Roberto Ranon, Marc Christie, Tommaso Urli. Accurately Measuring the Satisfaction of Visual Properties in Virtual Camera Control. Smart Graphics, 10th International Symposium on Smart Graphics, Jun 2010, Banff, Canada. pp.91-102, 10.1007/978-3-642-13544-6_9 . inria-00540505

HAL Id: inria-00540505

https://inria.hal.science/inria-00540505v1

Submitted on 29 Nov 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Accurately Measuring the Satisfaction of Visual Properties in Virtual Camera Control

Roberto Ranon[1], Marc Christie[2], and Tommaso Urli[1]

[1] HCI Lab, University of Udine, via delle Scienze 206, 33100, Udine, Italy
[2] IRISA/INRIA Rennes Bretagne Atlantique, Campus de Beaulieu, 35042, Rennes Cedex, France

Abstract. Declarative approaches to camera control model inputs as properties on the camera and then rely on constraint-based and/or optimization techniques to compute the camera parameters or paths that best satisfy those properties. To reach acceptable performances, such approaches often (if not always) compute properties satisfaction in an approximate way. Therefore, it is difficult to measure results in terms of accuracy, and also compare approaches that use different approximations. In this paper, we propose a simple language which can be used to express most of the properties proposed in the literature and whose semantics provide a way to accurately measure their satisfaction. The language can be used for several purposes, for example to measure how accurate a specific approach is and to compare two distinct approaches in terms of accuracy.

1 Introduction

Camera Control is an essential component of a large range of 3D applications, including 3D games, data exploration and visualization, virtual walk-throughs, 3D modelling and virtual storytelling [9]. In the literature, a number of camera control techniques have been proposed, ranging from interactive techniques – where the user directly or indirectly controls the camera parameters– to automated approaches in which that camera parameters and paths are computed automatically in a way that supports the user in the realisation of his tasks.

Within the category of automated approaches, *declarative* approaches focus on providing a general solution to the problem of camera control by following a three-step process: first design a general language to model camera control problems, then provide techniques to solve the problems described by the language, and finally propose means to explore the classes of solutions. Such declarative approaches generally model camera control problems as a set of properties that need to hold (*i.e.* constraints to solve) and a set of properties that should hold whenever possible (*i.e.* cost functions to optimize). The literature on declarative approaches to camera control reports three classes of properties:

– properties that directly bind camera parameters/path parameters to given values *e.g.* fix the camera up vector to ensure a horizontal view;

- properties that express geometric requirements on camera parameters/paths with respect to objects in the scene, *e.g.* requiring that the camera be at a certain distance from an object;
- properties that express requirements on the image(s) the camera will generate, *e.g.* requiring a certain object to be positioned in a given portion of the screen and not be occluded.

The last category of properties (hereinafter referred to as *screen-space properties*) is particularly interesting for several reasons. First, screen-space properties allow the user (or the application) to reason in terms of the result of the camera control process, rather than in terms of values for camera parameters or camera paths. Second, these are closer to the language a photographer/cinematographer would use to position a camera in a real world. For example, the approach described by Christie and Normand [8] uses properties such as *Close-up* and *Long shot* to express requirements on the size of objects in images. Third, they represent an expressive groundwork on which to build more abstract properties for example derived from cinematography and photography (e.g. rules of the thirds; Gooch *et al.* [11] use this rule to improve the composition of shots through local modifications of the camera parameters).

An overlooked issue in approaches to declarative camera control is accuracy: how exact is the evaluation of a shot, with regard to a set of properties. To reach acceptable computing times, declarative approaches often (if not always) compute satisfaction of screen-space properties in an approximate way or using approximate models, therefore leading to situations where the layout of objects on screen does not fully satisfy the given specification (e.g. an object may be evaluated as fully occluded while it is only partially occluded). Approximate models and approximate evaluations lead to both *false-good* and *false-bad* cases. A contribution may then consist in designing exact methods to accurately evaluate camera configurations with regard to screen-space properties, thereby offering means to compare the quality of different approaches, and to precisely evaluate the loss in accuracy of approximate models and approximate evaluators. The difficulty of reasoning about the efficiency/accuracy tradeoff is, in our view, one of the factors that limits the development of better declarative camera control approaches.

This papers deals with the above issues in two ways:

- it proposes a simple language which can be used to write expressions that capture most of the screen-space properties proposed in the literature, with the aim of providing a way to clearly express the meaning of screen-space properties;
- the language semantics provides a way to accurately measure if a camera satisfies an expression written in the language, thus allowing one to reason about the accuracy of a solution provided by existing declarative camera control approaches, or compare solutions together. However, while in principle our evaluation method could be used in place of approximate techniques in declarative camera control approaches developed so far, it is currently

too computational costly for that purpose (see discussion at the end of the paper).

In this paper, we restrict our proposal to static camera control problems, i.e. Virtual Camera Composition problems, where the task is to compute the optimal configuration of camera parameters given a set of properties that must hold in a given 3D scene at a certain instant in time. Extensions to dynamic situations (where the task is to find camera paths, or more generally how camera parameters must change in time) represents a far more complex task and will be considered in future work.

The paper is organized as follows: Section 2 provides an overview of screen-space properties proposed by approaches in the literature; Section 3 presents our language for camera control, while in Section 4 we show how to express screen-space properties from the literature using our language. In Section 5 we present an example where we evaluate the accuracy of an existing approach in the literature in a specific situation. Finally, in Section 6 we conclude the paper and mention future work.

2 Screen-space Properties in the Literature

Declarative approaches to camera control have been constantly exploring the balance between two orthogonal aspects: expressiveness and computational cost [9]. Expressiveness encompasses the range, nature and accuracy of constraints with which properties can be expressed over camera shots. For example, the classical `Framing` property constrains a target object to project inside a user-defined frame on the screen. The need for improving the expressiveness *i.e.* designing a cinematographic language that is both able to express a large range of properties, and that offers means to compose these properties, is of increasing importance with regard to the evolution of viewpoint computation and cinematography as demonstrated in most recent computer games [14]. On the other hand, the computational cost allotted to any camera control process directly drives the implementation of declarative approaches, generally at the cost of reducing the expressiveness. More importantly, the computational cost drives the level of abstraction of the geometry (how precisely should objects be represented and which geometric abstractions should be considered for the different properties). For example, a number of contributions rely on bounding representations for computing the visibility of targets, by casting rays from the camera to the vertices of the axis aligned bounding box [4, 5]. Furthermore limitations in the solving techniques restrict the range of possible abstractions (e.g. gradient-based techniques would require the evaluation of properties to be a differentiable function, or at least a smooth one [10]).

To provide the readers with an overview of screen-space properties used in declarative languages for camera control, we identified and gathered most common properties from the literature and display them in Table 1.

Property	Description		
Occlusion	Checks if target T is occluded on the screen	Occluded(T) [7], Occlusion Avoidance (T) [1]	Evaluation of occlusion is performed using projected bounding spheres.
	Checks if target T_2 occludes T_1	Occluded (T_1, T_2) [8] OccludedBy (T_1, T_2) [13]	Evaluation uses *occlusion cones* defined by the spherical bounds of the targets or hierarchical bounding spheres plus z-buffer.
	Checks if target T is not occluded on the screen	Occluded (T) [13]	Evaluation uses hierarchical bounding spheres plus z-buffer.
	Checks if no more than fraction min of the target T is occluded	OBJ_OCCLUSION_MINIMIZE (T, min) [2, 3]	Evaluation performed with nine-rays ray-casting to the bounding volume or off-screen rendering.
	Checks if a target T is out of view or occluded	OBJ_EXCLUDE_OR_HIDE(T) [2]	Evaluation not described in paper.
Framing	Checks whether a target T projects into a rectangular frame F on the screen	Framing(T,F) [7, 8]	Evaluation not described in papers.
	Checks whether target T projects into intervals X and Y	CenterXY(T,X,Y) [13]	Evaluation not described in paper.
	Checks whether horizontal/vertical orientation of target T projects into intervals X and Y	AngleXY(T,X,Y) [13]	Evaluation not described in paper.
	Checks whether target T is location (r)ight, (l)eft or (c)enter on the screen	Placement(T,r|l|c) [4]	Evaluation not described in paper.
Screen Separation	Checks whether a target T_1 is at a distance d to target T_2 on the screen	ScreenSeparation(T_1,T_2,d) [7, 13]	Evaluated as distance between the projected centers, minus the projected radiuses of the objects.
In Field Of View	Checks whether a target T is projected in the field of view	OBJ_IN_FIELD_OF_VIEW(T) [2, 3], InViewVolume(T) [5, 6], InView(T) [4], InViewport(T) [13]	Evaluated by projecting the bounding box or locus sphere of the object on the screen and checking if it is completely enclosed.
	Checks if a target T is out of view or Occluded	OBJ_EXCLUDE_OR_HIDE(T) [2]	Evaluation not described in paper.
	Checks if a target T is excluded from the field of view	Excluded(T) [4]	Evaluation not described in paper.
Size	Checks the projected size of a target T is in a given range v={close-up, medium close-up, long-shot...}	Projection(T,v) [8]	The distance is estimated from the size of the projected object: $distance = (\frac{sizeObject}{screenSize}) \cdot (\frac{1}{tan(fov/2)})$. An epsilon is used to shift the scale shots.
	Checks the area of a projected target T is equal to a value v	ProjectedArea(T,v) [13] Size(T,v) [4, 5, 12], ProjectionSize(T,v) [6], OBJ_PROJECTION_SIZE(T,v) [2, 3]	Evaluation uses the projected radius of the bounding sphere, the longest edge of the bounding box, the convex hull of the projected bounding box or off-screen rendering.
Relative Spatial Location	Checks whether projected target T_1 is (l)eft/(r)ight/(a)bove/(b)elow projected target T_2	RelativePositioning (T_1,T_2,l|r|a|b) [3, 8]	Evaluation done as geometric reasoning on the projected bounding volume of the objects or considering the position of the camera with respect to a plane which passes through the centers of the objects.
	Checks whether projected target T_1 is between projected targets T_2 and T_3 in X or Y dimensions	BetweenObjects(T_1,T_2, T_3,X|Y) [13]	Evaluation not explained in paper.

Table 1: Main screen-space properties from the literature.

For a thorough overview of solving techniques in virtual camera control, we refer the readers to the overview proposed in [9].

3 The Camera Evaluation Language

In this Section, we propose the *Camera Evaluation Language* (hereinafter, *CEL*). Its design is guided by two motivations: first, to propose a simple language that can be used by present and future declarative approaches to express their screen-space properties, and then to provide a way to reason about accuracy and the approximations that eventually need to be included. As we will see, all screen-space properties in table 1 can be expressed with a few simple CEL primitives, namely the *Rendering* and *Pixel Set* operators, and the common mathematical, logical and set operators and relations.

Operator or Relation	Returns
R(subscene)	PS = the set of pixels p that results from rendering *subscene* from *camera*, with $p_{side} = 0$ for each p
CR(subscene)	PS = the set of pixels p that results from rendering *subscene* from the position of *camera*, using 90 degrees FOV, perspective projection and view direction towards any face of a cube centered in the camera, with $p_{side} = 0, \ldots, 5$ depending on which side of the cube p belongs to
$Max_x(PS, side)$ Min_x, Max_y, Min_y Max_z, Min_z	$max(\{p_x \mid p \in PS \wedge p_{side} = side\})$ ... ...
$Avg_x(PS)$ Avg_y, Avg_z	$avg(\{p_x \mid p \in PS\})$ ...
$Overlap(PS_1, PS_2)$	set of pixels in PS_1 that have the same $x, y, side$ coordinates of some pixels in PS_2
$CoveredBy(PS_1, PS_2)$	set of pixels of PS_1 that would be covered by pixels of PS_2 if we rendered together the subscenes that produced PS_1 and PS_2
$Left(PS_1, PS_2)$ *Right, Above, Below*	set of pixels of PS_1 that are left of any pixel in PS_2, considering only pixels with $p_{side} = 0$ ...
$Distance(PS_1, PS_2)$	$min(distance(p, p')$ where $p \in PS_1, p' \in PS_2$, , considering only pixels with $p_{side} = 0$

Table 2: Operators and relations in CEL. In the table, PS, PS_1, PS_2 denote pixel sets.

3.1 Rendering Operators

Rendering operators take any subpart of a 3D scene (e.g., an object, a group of objects, or the entire scene), render it into an image with size $imageWidth$,

imageHeight (or a cube map), and return a set containing all pixels that, in the image, refer to the part of the 3D scene given as argument. Rendering operators are the primitive components of the language on which all operations are performed.

Rendering operators assume the existence of a current *camera*, from which rendering is performed. This is the camera we want to evaluate with respect to a set of properties. CEL defines two rendering operators, *R* and *CR* (see table 2, first two rows). The first one just returns the set of pixels resulting from rendering its argument into a 2D texture. The second one returns the set of pixel resulting from rendering its argument into a cube map from the *camera* position, using six orthogonal views (i.e., the operator arguments are rendered six times, one for each face of the cube map, starting from the current camera view direction).

Pixel sets that are returned by rendering operators are defined as follows: each pixel p is defined by its coordinates p_x, p_y, p_z, p_{side} where p_x, p_y are the coordinates of the pixel in the rendered image (or side of the cube map), p_z it its distance from *camera* and $p_{side} = 0, 1, \ldots, 5$ denotes one of the sides of the cube in case a cube map has been rendered, and is 0 in case we have rendered a 2D image. The concept of pixel set is similar to the notion of depth sprite or nailboard in image-based rendering [15].

At a practical level, the resulting pixel set can be easily computed by rendering the specified subscene part, and then take the resulting pixels (*i.e.* where color is different from the background, or the corresponding value of the Z-buffer is less than the maximum z value of the depth buffer).

3.2 Pixel Set Operators and Relations

Once the rendering is performed into pixel sets, a number of comparisons can be performed by applying simple operators over the pixel sets (*e.g.* computing the overlapping regions or relative spatial location). Such Pixel Set operators (see table 2, third to eighth row) and relations act on sets of pixels, perform calculations, and return numbers or pixel sets. In the following, $p(x, y, z, side) \in PS$ is true if there exists a pixel $p \in PS$ with such coordinates (similarly, we define also $p(x, y, side)$).

Besides the operators that perform basic calculations on one set (*Max*, *Avg* and *distance*, see Table 2, third and fourth row), we define the *Overlap* operator ($Overlap(PS_1, PS_2)$) that returns those pixels in PS_1 that have the same $x, y, side$ coordinates as pixels in PS_2 (see figure 1), i.e.:

$$Overlap(PS_1, PS_2) = \{p(x, y, side) \in PS_1 | p'(x, y, side) \in PS_2\}$$

Since we consider also the *side* coordinate in the operator, when two pixel sets resulting from *CR* are used, the comparison is done on each of the six generated images.

The operator $CoveredBy(PS_1, PS_2)$ returns the pixels in PS_1 that would be discarded by z-test if we rendered together the subscenes that produced PS_1 and

PS_2, i.e. it takes into account the overlapping region, and perform comparison on the z coordinate:

$$CoveredBy(PS_1, PS_2) = \{p(x, y, side) \in PS_1 | p'(x, y, side) \in PS_2 \wedge p_z > p'_z\}$$

The following operators act only on pixels with $p_{side} = 0$. More specifically, $Left(PS_1, PS_2)$ returns the pixels in PS_1 that are left of any pixel in PS_2, i.e.

$$Left(PS_1, PS_2) = \{p \in PS_1 | p_x < Min_x(PS_2)\}$$

Similarly, we define also *Right*, *Above*, and *Below* (these last two perform the comparison using the y coordinate).

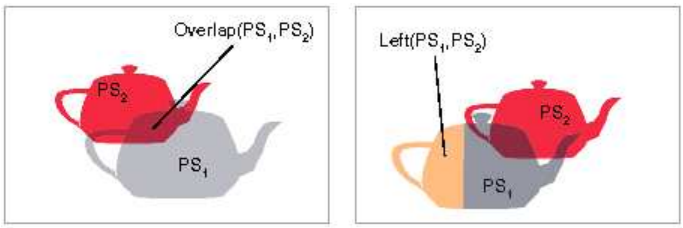

Fig. 1: The *overlap* operator (left) and the *Left* operator (right) in the case of pixel sets obtained by using the R operator.

4 Using CEL to build and evaluate screen-space properties

In this Section, we propose some CEL expressions to measure values related to the screen-space properties defined in the literature.

The following expressions express and evaluate the size of a target T relative to the viewport size, *i.e.* its height, width or area on the viewport:

$$\mathbf{Height}(T) = \frac{Max_y(R(T)) - Min_y(R(T))}{imageHeight}$$
$$\mathbf{Width}(T) = \frac{Max_x(R(T)) - Min_x(R(T))}{imageWidth}$$
$$\mathbf{Area}(T) = \frac{|R(T)|}{imageWidth \times imageHeight}$$

The following expressions express and evaluate the relative position on the viewport of two targets T_1, T_2:

$$\mathbf{ScreenSeparation}(T_1, T_2) = \frac{Distance(R(T_1), R(T_2))}{imageWidth}$$

$$\mathbf{LeftOf}(T_1, T_2) = \frac{|Left(R(T_1), R(T_2))|}{|R(T_1)|}$$

$$\mathbf{RightOf}(T_1, T_2) = \frac{|Right(R(T_1), R(T_2))|}{|R(T_1)|}$$

$$\mathbf{AboveOf}(T_1, T_2) = \frac{|Above(R(T_1), R(T_2))|}{|R(T_1)|}$$

$$\mathbf{BelowOf}(T_1, T_2) = \frac{|Below(R(T_1), R(T_2))|}{|R(T_1)|}$$

$$\mathbf{InFrontOf}(T_1, T_2) = \frac{|CoveredBy(Overlap(R(T_1), R(T_2)), R(T_1))|}{|Overlap(R(T_1), R(T_2))|}$$

$$\mathbf{InsideOf}(T_1, T_2) = \frac{|CoveredBy(R(T_2), R(T_1))|}{|R(T_1)|}$$

To express and evaluate inclusion of a target in the viewport, we use an additional geometry *CVV* (*Camera View Volume*) whose shape, position and orientation corresponds to the Camera View Volume:

$$\mathbf{InViewVolume}(T) = \frac{|CoveredBy(CR(T), CR(CVV))|}{|CR(T)|}$$

i.e., we compute the fraction of pixels of the target that are covered by pixels of the view volume. By using *CR*, we also take into account the pixels that are out of the viewport.

To express and evaluate framing, we use a similar idea, i.e. we render an additional geometry *SAS* (*Screen Aligned Shape*) which is a 2D shape positioned just after the camera near plane):

$$\mathbf{Framing}(T, SAS) = \frac{|CoveredBy(CR(T), CR(SAS))|}{|CR(T)|}$$

i.e., we compute the fraction of pixels of the targeted that are covered by pixels of the view volume. This approach allows to use any shape as the frame, and also, since we use *CR*, to set the frame (partly) outside the viewport.

Finally, occlusion can be expressed and measured by the following expressions (*Scene* denotes the entire scene):

$$\mathbf{Occluded}(\mathbf{T}) = \frac{|CoveredBy(R(T), R(Scene - T))|}{|R(T)|}$$

$$\mathbf{OccludedBy}(\mathbf{T_1}, \mathbf{T_2}) = \frac{|CoveredBy(R(T_1), R(T_2))|}{|R(T_1)|}$$

In cases where, for some argument T of an operator, *R(T)*=$\emptyset$ (i.e. the argument is not in the camera view volume), there might be two problems:

– the expression computes a division by zero (e.g. *Occluded*), or
– the expression does not know how to compute a result (e.g. *ScreenSeparation*).

In those cases, we define the expression to return -1 as a result. Note that this is not the case of *Height*, *Width* and *Area*, where if the target is not in the camera view volume, the computed result (i.e. zero) is correct.

The accuracy of evaluating an expression depends on the size of the image to which rendering operators compute their results. Of course, since these operators are based on rasterization, they can never be perfectly correct. However, we can safely assume the evaluation as accurate when rendering to images that are at least the same size as the intended application window, since using a greater accuracy would not be appreciable by the user.

Since screen-space properties in the literature typically are expressed with respect to some desired value (e.g. height of an object equal to ...), in order to express screen-space properties we define also a comparison function:

$$Equal(expr_1, expr_2, v_0, v_1, bl) = \begin{cases} 0 \text{ if } expr_1 \notin [v_0, v_1] \vee expr_2 \notin [v_0, v_1] \\ 1 + |expr_1 - expr_2| \dfrac{bl - 1}{v_1 - v_0} \text{ otherwise} \end{cases}$$

where $expr_1$, $expr_2$ are two CEL expressions (or possibly, just a numeric value), $0 \leq v_0 < v_1$ define an interval inside which the comparison is at least partly satisfied, and $0 \geq bl \geq 1$ defines the minimum value of satisfaction when both $expr_1$, $expr_2$ are in $[v_0, v_1]$. *Equal* returns a value in [0,1] indicating how close the actual values computed for the expressions are, provided that they are inside the acceptable range (if not, the returned value is zero):

Figure 2 shows the *Equal* function with different *baseline* settings.

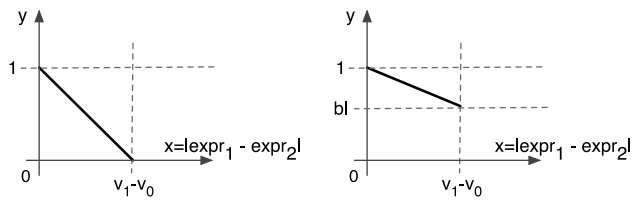

Fig. 2: The *Equal* function with bl=0 (left), bl=0.6 (right). To simplify the graph, we have represented $|expr_1 - expr_2|$ in the horizontal axis

For example, $Equal(OccludedBy(T_1, T_2), 0.7, 0.5, 1, 0)$ means that we would like T_1 occluded by T_2 for the 70% of its rendered area. Satisfaction will be therefore 1 for that value. If the occluded fraction is less than 0.5, satisfaction will be 0. For values between 0.5 and 0.7, satisfaction will vary in [0,1] and be proportional to how close the value is to the desired 0.7 value.

5 Example: using CEL to measure the accuracy of existing approaches

In this Section, we show how to use CEL to measure the accuracy of the approaches proposed by Burelli et al. [5] and Christie et al. [8] for a specific situation, i.e. the camera from which the picture in figure 3 has been rendered. The set of screen-space properties we consider in this example are (in the formulation adopted in [5]) listed in the first column of Table 3 (t is the transporter in figure 3, which shows also its bounding box); their CEL equivalent expressions are shown in the third column of the same table.

Fig. 3: The image computed by the camera we are evaluating with respect to the properties in Table 3

The first property requests the transporter to be entirely in the camera view volume. In [5], the paper abstracts the transporter using an Axis Aligned Bounding Box (AABB) representation, it returns 1 if all the corners of its AABB are in the camera view volume, 0 if no corner is in the camera view volume, and 0.5 otherwise. Thus the value computed by [5] is 0.5 (partially in view volume) whereas the target is fully included in the view volume. The second property requires the transporter to be fully unoccluded. Since [5] measures that by ray casts towards the corners and center of the transporter AABB, it reports the object to by fully unoccluded which is obviously false. The third property requires the transporter projected size to be 30% of the image area. Since [5] measures size by evaluating the area of the projected bounding sphere, it returns the value 0.88 (i.e. we are 88% close to the desired value). Now, the fourth column of Table 3 reports the values obtained by accurate evaluation using the equivalent CEL expressions. The total error accumulated by [5] is 1.33, *i.e.* the approach is off by an average of 44% compared to accurate evaluation, and the worst approximation is given by the *objProjSize* property. Now, we compare the results with Christie & Normand [8]. The authors rely on a spherical representation to abstract targets and

since the transporter is a long shaped object, the Framing property returns a very approximate result (only 37% is in the frame). The NoOcclusion studies the overlap of the projecting spheres of the target and of the occluders and clearly states that the target is occluded by a large value (the bounding of the lamp is very large). Since the projection size is not modeled in [8], we replace it by a Long Shot property. Average mistake in the case of paper [8] is 68%.

Property (as in [5])	value (as in [5])	CEL expression	CEL value	Error
$objInFOV(t, 1.0, 1.0)$	0.5	$\mathbf{InViewVolume}(t) = 1$	1.0	50%
$objOcclusion(t, 0.0, 1.0)$	1.0	$\mathbf{Occluded}(t) = 0$	0.87	13%
$objProjSize(t, 0.3, 1.0)$	0.88	$\mathbf{Area}(t) = 0.3$	0.18	70%
Property (as in [8])	value (as in [8])	CEL expression	CEL value	Error
$Framing(t, -1, 1, -1, 1)$	0.37	$\mathbf{InViewVolume}(t) = 1$	1.0	63%
$NoOcclusion(t)$	0.1	$\mathbf{Occluded}(t) = 0$	0.87	90%
$LongShot(t)$	0.7	$\mathbf{Area}(t) = 0.3$	0.18	52%

Table 3: Properties and their measured values for the viewpoint displayed in Figure 3; t is the transporter.

6 Discussion and Conclusions

In this paper, we have presented a simple but expressive language for specifying screen-space properties in Virtual Camera Control problems. The language enables a clear specification of most properties in the literature and furthermore enables the accurate measuring of their satisfaction. We hope that the language can also be a valuable tool in measuring accuracy when designing and implementing new camera control approaches where complex geometries need to be abstracted and approximations performed for the sake of performance. To this purpose, an implementation of the language, as well as more examples of using it are available at `http://www.cameracontrol.org/language`.

Future work will consist in extending the language to deal with dynamic camera control, and use it in the direction of establishing a benchmarking environment to compare models, techniques and algorithms in Virtual Camera Control. Moreover, we are also exploring the possibility of computing CEL expressions in the GPU, together with low resolution pixel sets, and see if we can get performances that are acceptable to be used inside a declarative camera control approach.

Acknowledgments Authors acknowledge the financial support of the Italian Ministry of Education, University and Research (MIUR) within the FIRB project number RBIN04M8S8, as well as European Grant number 231824.

References

[1] W. H. Bares and J. C. Lester. Intelligent multi-shot visualization interfaces for dynamic 3d worlds. In *IUI '99: Proceedings of the 4th international conference on Intelligent user interfaces*, pages 119–126. ACM, New York, NY, USA, 1999.

[2] W. H. Bares, S. McDermott, C. Boudreaux, and S. Thainimit. Virtual 3d camera composition from frame constraints. In *MULTIMEDIA '00: Proceedings of the eighth ACM international conference on Multimedia*, pages 177–186. ACM, New York, NY, USA, 2000.

[3] W. H. Bares, S. Thainimit, and S. McDermott. A model for constraint-based camera planning. In *Proceedings of AAAI Spring Symposium on Smart Graphics*, pages 84–91. 2000.

[4] O. Bourne, A. Sattar, and S. Goodwin. A constraint-based autonomous 3d camera system. *Constraints*, 13(1-2):180–205, 2008. ISSN 1383-7133. doi:http://dx.doi.org/10.1007/s10601-007-9026-8.

[5] P. Burelli, L. Di Gaspero, A. Ermetici, and R. Ranon. Virtual camera composition with particle swarm optimization. In *SG '08: Proceedings of the 9th international symposium on Smart Graphics*, pages 130–141. Springer-Verlag, Berlin, Heidelberg, 2008.

[6] P. Burelli and A. Jhala. Dynamic artificial potential fields for autonomous camera control in 3d environments. In *Proceedings of Artificial Intelligence and Interactive Digital Entertainment (AIIDE09)*. AAAI Press, 2009.

[7] M. Christie and E. Languénou. A constraint-based approach to camera path planning. In *Smart Graphics*, pages 172–181. 2003.

[8] M. Christie and J.-M. Normand. A semantic space partitionning approach to virtual camera control. In *Proceedings of the Annual Eurographics Conference*, pages 247–256. 2005.

[9] M. Christie, P. Olivier, and J.-M. Normand. Camera control in computer graphics. *Comput. Graph. Forum*, 27(8):2197–2218, 2008.

[10] S. M. Drucker and D. Zeltzer. Intelligent camera control in a virtual environment. In *Proceedings of Graphics Interface 94*, pages 190–199. 1994.

[11] B. Gooch, E. Reinhard, C. Moulding, and P. Shirley. Artistic composition for image creation. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 83–88. Springer-Verlag, London, UK, 2001. ISBN 3-211-83709-4.

[12] N. Halper, R. Helbing, and T. Strothotte. A camera engine for computer games: Managing the trade-off between constraint satisfaction and frame coherence. *Comput. Graph. Forum*, 20(3), 2001.

[13] N. Halper and P. Olivier. CamPlan: A camera planning agent. In *Smart Graphics 2000 AAAI Spring Symposium*, pages 92–100. AAAI Press, 2000.

[14] B. Hawkins. *Real-Time Cinematography for Games (Game Development Series)*. Charles River Media, Inc., Rockland, MA, USA, 2004. ISBN 1584503084.

[15] G. Schaufler. Nailboards: A rendering primitive for image caching in dynamic scenes. In *Rendering Techniques 97: Proceedings of the Eurographics Rendering Workshop*, pages 151–162. Springer Verlag, 1997.