Virtual Camera Composition with Particle Swarm Optimization

Paolo Burelli[1], Luca Di Gaspero[2], Andrea Ermetici[1], and Roberto Ranon[1]

[1] HCI Lab, University of Udine, via delle Scienze 206, 33100, Udine, Italy
roberto.ranon@dimi.uniud.it
[2] DIEGM, University of Udine, via delle Scienze 208, 33100, Udine, Italy
l.digaspero@uniud.it

Abstract. The Virtual Camera Composition (VCC) problem consists in automatically positioning a camera in a virtual world, such that the resulting image satisfies a set of visual cinematographic properties [3]. In this paper, we propose an approach to VCC based on *Particle Swarm Optimization* [5]. We show, in realistic situations, that our approach outperforms a discretized, exhaustive search method similar to a proposal by Bares et al [1].

1 Introduction

In 3D graphics interactive applications, effective camera placement and control is fundamental for the user to understand the virtual environment and be able to effectively accomplish the intended task. For example, in 3D information visualization, bad camera placements could cause the user to miss important visual details (e.g., because they are occluded) and thus make wrong assumptions on the data under analysis.

In most current 3D applications, users directly position the camera using a input device through a tedious and time-consuming process requiring a succession of "place the camera" and "check the result" operations [3]. In recent years, some researchers (e.g., [1, 3, 6, 8]) have come up with methods to automatically position the camera that aim at relieving the user from direct control, and are inspired by how human cinematographers approach the same problem: first, requirements on the image or shot to be obtained are stated (e.g., by the user), and then, a camera fulfilling those requirements is computed.

More specifically, the Virtual Camera Composition (VCC) problem consists in positioning a camera in a virtual world, such that the resulting image satisfies a set of visual cinematographic properties [3], e.g. subjects' size and location. The approaches developed so far typically model VCC as a constraint satisfaction or optimization problem (some approaches use both) where the desired image properties are represented as constraints or objective functions. A range of different solving techniques [4] have been explored in the past, but generating effective results in real-time (or near-real time), even with static scenes, remains an issue. Since this requirement is very important in interactive applications, there is the need of finding more efficient methods, as well as to compare the performances of previously proposed approaches in realistic contexts.

In this paper, we present and evaluate an approach to VCC that employs *Particle Swarm Optimization* (hereinafter, PSO) [5], a method which, to the best of our

knowledge, has never been applied to this kind of problems. PSO is a population-based method for global optimization, which is inspired by the social behavior that underlies the movements of a swarm of insects or a flock of birds. We will show how the PSO approach can be used to solve VCC problems in the case of static scenes, and evaluate its performances against a discretized, exhaustive search approach very similar to the one proposed by Bares et al in [1]. Finally, one of our motivations for this research is to use automatic cameras to support users' navigation in virtual environments. Although this is not the focus of this paper, we will briefly present the idea in our experimentation.

The paper is organized as follows. Section 2 reviews related work, while Section 3 describes our approach to VCC. Section 4 presents the experimental results. Finally, in Section 5 we conclude the paper and outline future work.

2 Related Work

A comprehensive survey of approaches to camera control can be found in [4]. In the following, we focus on approaches to VCC that: (i) employ a declarative "cinematographic" style, i.e. where the problem is expressed as a set of requirements on the image computed from the camera, such as relative viewing angles and occlusions, and (ii) model the problem as a constraint and/or optimization system. These approaches are by far the most general, and therefore interesting for a wide range of applications.

In constrained search and/or optimization approaches to VCC, the properties of the image computed from the camera are expressed as numerical constraints on the camera parameters, (typically, camera position, orientation, and FOV). Although pure optimization (e.g., [6]) or constraint satisfaction (e.g., [2]) have been used in the past, more recent proposals tend to adopt an hybrid strategy, where some requirements are modeled as constraints, and used in a first phase to compute geometric volumes of feasible camera positions [1, 3, 8]. Then, in a second phase, requirements are modeled as objective functions that are maximized by searching inside the geometric volumes using various optimization methods, such as stochastic [3, 8] or heuristic [1] search.

The advantage of the hybrid approach is that it can reduce complexity by limiting the search space using geometric operators to implement constraints in the first phase, and, at the same time, the optimization phase can increase the chances (with respect to a pure constraint-based approach) that a (possibly good) solution will be found: in many situations, it is better to have a solution, although not satisfying some requirements, than having no solution at all. Another advantage is the fact that the geometric volumes generated in the first phase can be semantically characterized with respect to their visual properties [3], e.g. to allow the computation of multiple, potentially equivalent solutions instead of just generating a single one.

3 A PSO Approach to VCC

In this Section, we describe how PSO can be used to solve the VCC problem. First, we will present the language that can be used to describe the properties of the image to be generated from the camera. Our language comprises most of the visual properties from prior work including [1, 3], so, for reasons of space, we refer the reader to those papers

for a more detailed explanation. Then, we will describe the solving process. As other approaches mentioned in the previous section, we follow a hybrid strategy. Therefore, we first compute volumes of feasible camera positions from some of the requirements that constrain the position of the camera. This phase of the solving process is quite similar to [3] (our proposal can thus be considered a variation of that approach), so we will just give an high-level overview. The search inside the feasible regions is then carried out with PSO, on which we will focus in detail. As in all other approaches to VCC, we consider a classical pinhole Euler-based camera model where the parameters are the camera position, orientation, and FOV. Finally, the visual properties we adopt (as well as the solving method) are not meant for dynamic scenes with temporary occlusions, which requires properties expressed over more than just the current point in time.

3.1 Available Image Properties

The following is the list of properties that involve an object in the scene. Most properties include a real argument, w, whose value encodes the importance of the requirement.

- **Object view angle**. Requires the camera to lie at a specified orientation relative to an object: $objHAngle(\mathsf{Object}\, x, \mathsf{angle}\, \theta, \mathsf{angle}\, \gamma, \mathsf{double}\, w)$, where θ is the angle between the object front vector and the preferred viewing direction, and γ defines a range of allowed angles around the preferred direction; $objVAngle(\mathsf{Object}\, x, \mathsf{angle}\, \theta, \mathsf{angle}\, \gamma, \mathsf{double}\, w)$, where θ is the angle between the object up vector and the preferred viewing direction, while γ defines a range of allowed angles around the preferred direction;
- **Object inclusion in the image**. Requires a specified fraction $f \in [0, 1]$ of the object to lay inside the FOV of the camera: $objInFOV(\mathsf{Object}\, x, \mathsf{double}\, f, \mathsf{double}\, w)$ ($f = 0$ means the object must not be in the camera FOV);
- **Object projection Size**. Requires the projection of the object to cover a specified fraction $f \in]0, 1]$ of the image: $objProjSize(\mathsf{Object}\, x, \mathsf{double}\, f, \mathsf{double}\, w)$;
- **Object distance from camera**. Requires the object to lie at a specified distance from the camera: $objDistanceFromCam(\mathsf{Object}\, x, \mathsf{double}\, d_{min}, \mathsf{double}\, d_{max})$;
- **Object Position in Frame**. Requires a specified fraction $f \in [0, 1]$ of the object to lie inside a given rectangular subregion of the image: $objProjPosition(\mathsf{Object}\, x, \mathsf{2DPoint}\, p_1, \mathsf{2DPoint}\, p_2, \mathsf{double}\, f, \mathsf{double}\, w)$, where p_1, p_2 are two points in the image identifying the top-left and bottom-right corners of the rectangular region (0 means the object must not be inside the rectangle);
- **Object Occlusion**. Requires the specified fraction $f \in [0, 1]$ of the object projection to be occluded in the image: $objOcclusion(\mathsf{Object}\, x, \mathsf{double}\, f, \mathsf{double}\, w)$ (0 means the object should be not occluded at all).

Additionally, a set of camera-related properties are defined. They are useful to directly limit the search space when suitable; for example, upside-down cameras and cameras placed inside walls or other objects are typically unsuitable.

- **Camera outside region**. Requires the camera to lie outside a box-shaped region in 3D space: $camOutsideRegion(\mathsf{3DPoint}\, p_1, \mathsf{3DPoint}\, p_2)$, where p_1, p_2 are two opposite corners of the box.

– **Camera outside object**. Requires the camera to lie outside the bounding box of a specified object: $camOutsideObj(\mathsf{Object}\, x)$.
– **Camera above plane**. Requires the camera to lie above a given plane in 3D space: $camAbovePlane(\mathsf{3DPoint}\, o, \mathsf{3DPoint}\, n)$, where o is a point on the plane and n is the normal of the plane.
– **Bind camera parameters:** $camBindX(\mathsf{double}\, x_{min}, \mathsf{double}\, x_{max})$, $camBindY(\mathsf{double}\, y_{min}, \mathsf{double}\, y_{max})$, $camBindZ(\mathsf{double}\, z_{min}, \mathsf{double}\, z_{max})$ $camBindRoll(\mathsf{angle}\, \alpha_{min}, \mathsf{angle}\, \alpha_{max},)$, $camBindYaw(\mathsf{angle}\, \alpha_{min}, \mathsf{angle}\, \alpha_{max},)$, $camBindPitch(\mathsf{angle}\, \alpha_{min}, \mathsf{angle}\, \alpha_{max},)$, $camBindFOV(\mathsf{angle}\, \alpha_{min}, \mathsf{angle}\, \alpha_{max},)$, $camAspectRatio(\mathsf{double}\, f)$.

All these atomic properties can be combined to form more complex descriptions by using the logic operator $\wedge$.

3.2 Phase 1: Computing Volumes of Feasible Camera Positions

In this phase we use some of the specified properties as geometric operators to derive (possibly non connected) volumes in 3D space where it is feasible to position the camera. More particularly, the properties which are considered in this phase are:

– object-related properties: **Object View Angle**, **Distance from Camera**;
– camera-related properties: **Camera outside region**, **Camera outside object**, **Camera above plane** and **Bind camera parameters**.

For example, a **Distance from camera** property defines a feasible volume which is the difference of two spheres, whose center is the center of the bounding volume of the considered object, and whose radii are respectively the maximum and minimum allowable distances. The geometric operations for the other properties are defined in detail in [1, 3]. Note that, contrary to [3], we do not employ occlusion-related properties in this step because we prefer to avoid cutting too much the search space (with the risk of not generating any solution). The feasible volumes defined by each property are then combined with intersection operators to derive the (possibly non-connected) feasible volume into which the search with PSO will be carried out.

Technically, this phase has been implemented using the *VTK* library (`www.vtk.org`) that provides the required geometric primitives (planes, boxes, ...) and operators (intersection, difference, ...) to derive the feasible volumes of space as implicit functions. The optimization phase will then use the computed implicit functions to evaluate when a point lies inside or outside the feasible volume.

3.3 Phase 2: Searching inside the Feasible Volume with PSO

Swarm Intelligence is an Artificial Intelligence paradigm, which relies on the exploitation of the (simulated) behavior of self-organizing agents for tackling complex control and optimization problems. In particular, PSO [5] is a population-based method for global optimization, whose dynamics is inspired by the social behavior that underlies the movements of a swarm of insects or a flock of birds. These movements are directed

toward both the best solution to the optimization problem found by each individual and the global best solution.

More formally, given a D-dimensional (compact) search space $S \in \mathbb{R}^D$ and a scalar objective function $f : S \to \mathbb{R}$ that assesses the quality of each point $x \in S$ and has to be maximized, a *swarm* is made up of a set of N particles, which are located in that space. The i-th particle is described by three D-dimensional vectors, namely:

- the particle current *position* $\mathbf{x}_i = (x_{i_1}, x_{i_2}, \ldots, x_{i_D})$;
- the particle *velocity* $\mathbf{v}_i = (v_{i_1}, v_{i_2}, \ldots, v_{i_D})$, i.e., the way the particle moves in the search space;
- the particle *best visited position* (as measured by the objective function f) $\mathbf{P}_i = (p_{i_1}, p_{i_2}, \ldots, p_{i_D})$, a memory of the best positions ever visited during the search.

The index of the particle that reached the global best visited position is denoted by g, that is, $g = \arg\max_{i=1,\ldots,N} F(\mathbf{P}_i)$.

At the beginning of the search (step $n = 0$), the particles are set at random locations and with random velocities. The search is performed as an iterative process, which at step n modifies the velocity and position vectors of each particle on the basis of the values at step $n-1$. The process evolves according to the following rules (superscripts denote the iteration number):

$$\mathbf{v}_i^n = w^{n-1}\mathbf{v}_i^{n-1} + c_1 r_1 \left(\mathbf{p}_i^{n-1} - \mathbf{x}_i^{n-1}\right) + c_2 r_2 \left(\mathbf{p}_g^{n-1} - \mathbf{x}_i^{n-1}\right) \quad (1)$$

$$\mathbf{x}_i^n = \mathbf{x}_i^{n-1} + \mathbf{v}_i^n \qquad i = 1, 2, \ldots, N \quad (2)$$

In these equations, the values r_1 and r_2 are two uniformly distributed random numbers in the $[0, 1]$ range, whose purpose is to maintain population diversity. Constants c_1 and c_2 are respectively the so-called *cognitive* and *social* parameters, which are related to the speed of convergence. The value w^n is an *inertia weight* and it establishes the influence of the search history on the current move. A high weight is related to a global exploration, while a low weight allows a local exploration (also called exploitation). In Section 4, we will discuss the values we have chosen for all the parameters (including the number of particles and iterations).

Since at each iteration a solution to the problem is available (although it could not be the optimal one), PSO belongs to the family of *anytime* algorithms, which can be interrupted at any moment still providing a solution. In the general case, however, the iterative process is run until either a pre-specified maximum number of iterations has elapsed or the method has converged (i.e., all velocities are almost zero).

Having overviewed how PSO works, we now describe how we use it for searching inside the feasible volumes computed in the previous phase. In our case, the PSO search space is composed by all camera parameters, so it is a subset of $\mathbb{R}^7$. In particular, each particle position in $\mathbb{R}^7$ completely defines a camera, i.e. it assigns a value to each of the 7 camera parameters (3 position coordinates, 3 orientation angles, and FOV angle):

- the first three dimensions (x_0, x_1, x_2) correspond to the camera position. These values will be kept inside the feasible volume during the optimization process;

– the second three dimensions (x_3, x_4, x_5) correspond to the camera orientation, each ranging from 0 to 2π;
– the last dimension (x_7) corresponds to the camera FOV, ranging from a minimum to a maximum value;

In other words, PSO will optimize all camera parameters, but, with respect to the camera position (first three dimensions of the search space), search will be restricted inside the feasible volume computed in the previous phase. In practice, to check if a particle is inside or outside the feasible volume, we simply use x_0, x_1, and x_2 as arguments to the implicit functions derived in phase 1.

At the beginning of the search, N particles are set at random locations inside the feasible volumes (i.e., we randomly generate particles until we obtain N of them inside the feasible volume). Since particles might exit the feasible volume during the execution of PSO, after each iteration we check each particle with respect to the implicit functions (i.e., we check the position of the camera defined by the particle) and, if the particle is not in the feasible volume, we assign it the worst possible value of the objective function (i.e., 0). This will make the particle return into the feasible volume in the next iterations.

The objective function is built by taking into account the following object-related properties: **Object view angle**, **Object inclusion in the image**, **Object projection Size**, **Object Position in Frame**, and **Object Occlusion**. The degree of satisfaction of each property is evaluated by means of a function $f : \Pi \times S \rightarrow [0, 1]$ whose semantics depends on the property $\pi \in \Pi$ at hand. In general, the function measures a relative difference between the desired value for the property and its actual value. The value of f is then normalized in order to obtain a real value in the range $[0, 1]$, where 1 represents the satisfaction of the associated constraint and 0 is complete unfulfillment. For example, to evaluate an Object Position in Frame Property, we calculate the projection of the object bounding sphere, and then determine how much this projection covers the specified rectangular region. In the case of an occlusion property, we calculate the value of f by ray casting from the camera to the bounding box of the object (one to the center of the bounding box, and 8 to its corners), and testing intersections of the rays with other objects in the scene (where the number of rays intersecting other objects gives the percentage of occlusion). This is not optimal, since it might be problematic for large occluders and not very precise with respect to the expression of partial occlusion. However, how the satisfaction of each property is calculated is independent from PSO, and can thus be improved maintaining the general solving process. For the other properties, the calculation of the objective function is similar to the ones described in [1, 3, 4].

When atomic properties are combined by $\wedge$ connectors, the combined objective function is computed as $f(\pi_1 \wedge \pi_2, \mathbf{x}) = w_1 f(\pi_1, \mathbf{x}) + w_2 f(\pi_2, \mathbf{x})$, where the weights w_i are those given as last argument to each property.

In general, the evaluation of a particle requires the function f to be computed for all the properties of the image description. However, this process can be quite time consuming, especially in the case of complex image descriptions or with properties that require a computationally intensive evaluation (e.g., occlusion). Therefore, we adopt a *lazy* evaluation mechanism for the objective function: the computation of f for the particle i is stopped if the sum of the weights of the properties that still have to be evaluated is smaller than the best objective value $f(\pi, \mathbf{P}_i)$. Therefore, as a heuristic, it

could be useful to sort the properties leaving at the end the ones whose evaluation has a higher computational cost.

4 Implementation and experimental results

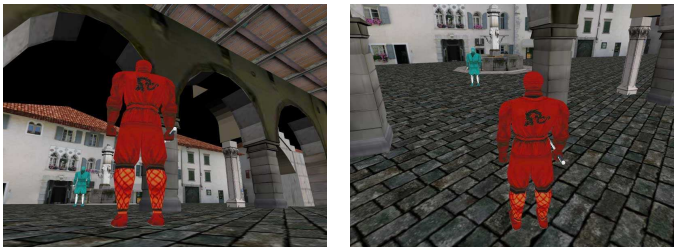
The VCC approach described in the previous section has been implemented as a part of a general C++ library which can be used in 3D applications. The library includes, besides the PSO solver, a discretized, exhaustive search-based algorithm (hereinafter called EXH) which follows the approach described in [1]. The first phase is identical for PSO and EXH. In the second phase, EXH scans a grid of $n \times m \times o$ points in the 3D scene (where n, m and o can be varied depending on the desired resolution). For each point in the grid, when it is inside the feasible volume, EXH considers a finite set of camera orientations (with differ by 15 degrees in each Euler angle) and FOV values, and for each one it evaluates the objective function exactly as PSO (including lazy evaluation). After having considered all points, or when the value of the objective function is sufficiently close to the optimum, EXH returns the best found camera.

In our experimentation, we will consider three increasingly complex VCC problems and show the resulting cameras generated by PSO and EXH. To make the test more realistic, we have set all the problems in the 3D model of a medieval village (which can be visited at `udine3d.uniud.it/venzone/en/index.html`), so that in evaluating occlusions the algorithms will have to take into account several objects.

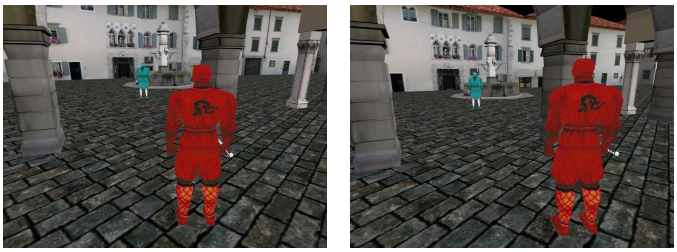
All the VCC problems have been solved 100 times each with both algorithms. Since the PSO approach can produce different cameras in different runs of the same problem, we will show more cameras for each problem solved by PSO, and just one camera for EXH (which calculates the same camera in each run). Then, we will discuss the computational performances. For PSO, we have employed in each problem a population of 64 particles and limited the maximum number of iterations to 50. These parameter values have been determined through experimentation, and have demonstrated to be fairly robust in all tests we have made. For the other parameters of PSO, we have employed $c_1 = c_2 = 0.5$ and an inertia weight w which linearly decreases from an initial value of 1.2 to 0.2, in order to balance between exploration and exploitation at different stages of the search. These values are considered a good choice in many problems to which PSO has been applied.

For EXH, we have employed in each test the smallest grid (in terms of number of points) such that the algorithm was able to generate a camera with fitness value close the one PSO was able to compute. Moreover, both approaches have been set to stop the search when they reach the threshold of 98% of optimal value. Both algorithms have been compiled with Microsoft Visual Studio 2005 C++ compiler and run on an Athlon 64 X2 5000+ (2.4 GHz), 4 GB ram running Microsoft Windows XP.

In each of the following tests, we have introduced a set of properties to limit camera orientation and FOV to suitable values. More specifically, we lock the up vector of the camera parallel to the world Y axis and the FOV to the value currently being used by the rendering engine, and limit the camera position inside a box containing the whole scene. The common properties used in the tests presented in this paper are listed the following (in all listings we omit the $\wedge$ connectors).



(a) (b)



(c) (d)

Fig. 1: Cameras computed from the properties listed in 1.2: (a) camera computed by EXH considering a $25 \times 25 \times 25$ grid of possible camera positions; (b,c,d) cameras computed by PSO in three different runs.

Listing 1.1: Properties common to all problems

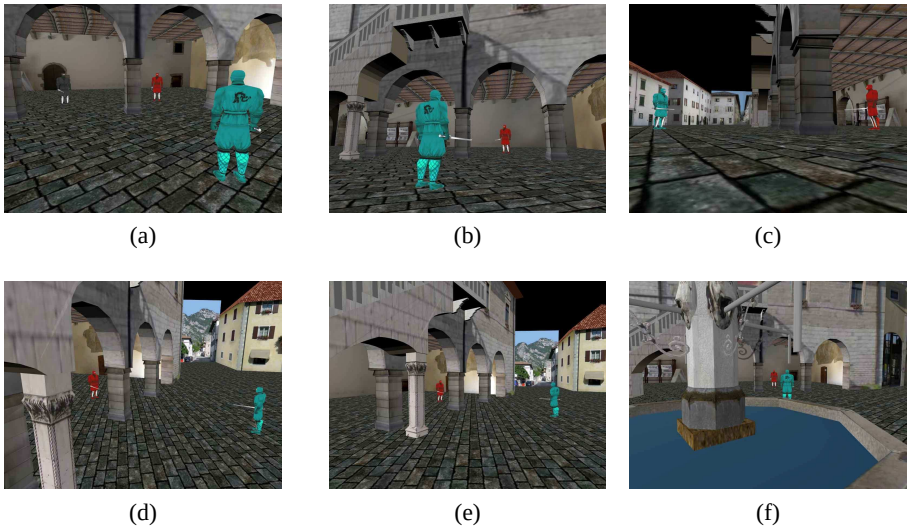
```
camBindRoll(0, 0)
camAspectRatio( currentScreenAspectRatio )
camBindFOV( currentCameraFOV )
camBindX( x_{min}, x_{max})
camBindY( y_{min}, y_{max})
camBindZ( z_{min}, z_{max})
```

Over the shoulder shot. The objective of this problem is to obtain a typical *over the shoulder* shot that shows the spatial relationship between two characters in the scene. The resulting cameras are shown in Figure 1.

Listing 1.2: Over the shoulder

```
camOutsideObject(blueWarrior)
objOcclusion(blueWarrior, 0.0, 1.0)
objInFOV(blueWarrior,1.0,1.0)
camOutsideObj(redWarrior)
objOcclusion(redWarior, 0.0, 1.0)
objInFOV(redWarrior,1.0,1.0)
objHAngle(redWarrior, -180, 90, 1.0)
objProjSize(redWarrior, 0.3, 1.0)
objVAngle(redWarrior, 45, 15, 1.0)
```

Occlusion. The objective of this problem is to obtain an image where one of three characters in the scene (whose relative positions are shown in Figure 2a) is completely



(a) (b) (c)

(d) (e) (f)

Fig. 2: Cameras computed from the properties listed in 1.3: (a) an image showing the relative position of three warriors in the scene; (b) camera computed by EXH considering a $10 \times 10 \times 10$ grid of possible camera positions; (c,d,e,f) cameras computed by PSO in four different runs.

occluded, while the others are completely visible. The resulting images are shown in Figure 2. Note that, in this case, the cameras produced by PSO (Figures 2c and 2d) in different runs are quite different.

Listing 1.3: Occlusion

```
camOutsideObject(blueWarrior)
objOcclusion(blueWarrior, 0.0, 1.0)
objInFOV(blueWarior,1.0,1.0)
camOutsideObject(redWarrior)
objOcclusion(redWarior, 0.0, 1.0)
objInFOV(redWarrior,1.0,1.0)
objOcclusion(blackWarrior, 1.0, 1.0)
```

Navigation aid. One interesting possibility is to use automatically generated cameras to help users in orienting themselves and in reaching places of interest in virtual environments. For example, by augmenting the scene with semantic information about landmarks, points of interest, and paths, one could derive generic rules that compute (and present to the user during navigation) cameras that highlight the most important or closer navigational elements. This could help users to: (i) learn landmarks during exploration and therefore increase the acquisition of navigational knowledge; (ii) see and recognize landmarks during orientation and search. By also including the user's avatar in the computed image, it is also possible to highlight the current spatial relation between the user's actual position and the navigational elements in the scene. The following problem regards a situation where we have some landmarks (the cathedral dome, its tower, the town hall tower and the baptistery) and we would like to compute

cameras that shows the avatar at a certain size (most important requirement, otherwise the user could not be able to find its position), plus most possible landmarks. The results obtained with different position of the user's avatar (which is the green character in the images) are reported in figure 3. Note that requests cannot be fully satisfied in any run, but nevertheless the computed cameras are at least able to highlight the user's position with respect to some of the landmarks.

Listing 1.4: Navigation Aid

```
objProjSize(avatar,0.1,5.0)
objOcclusion(avatar, 0.0, 10.0)
objInFOV(avatar,1.0,12.5)
objOcclusion(baptistery, 0.0, 1.0)
objInFOV(baptistery,1.0,1.0)
objOcclusion(town_hall_tower, 0.0, 1.0)
objInFOV(town_hall_tower,1.0,1.0)
objOcclusion(cathedral_dome, 0.0, 1.0)
objInFOV(cathedral_dome,1.0,1.0)
objOcclusion(church_tower, 0.0, 1.0)
objInFOV(church_tower,1.0,1.0)
```

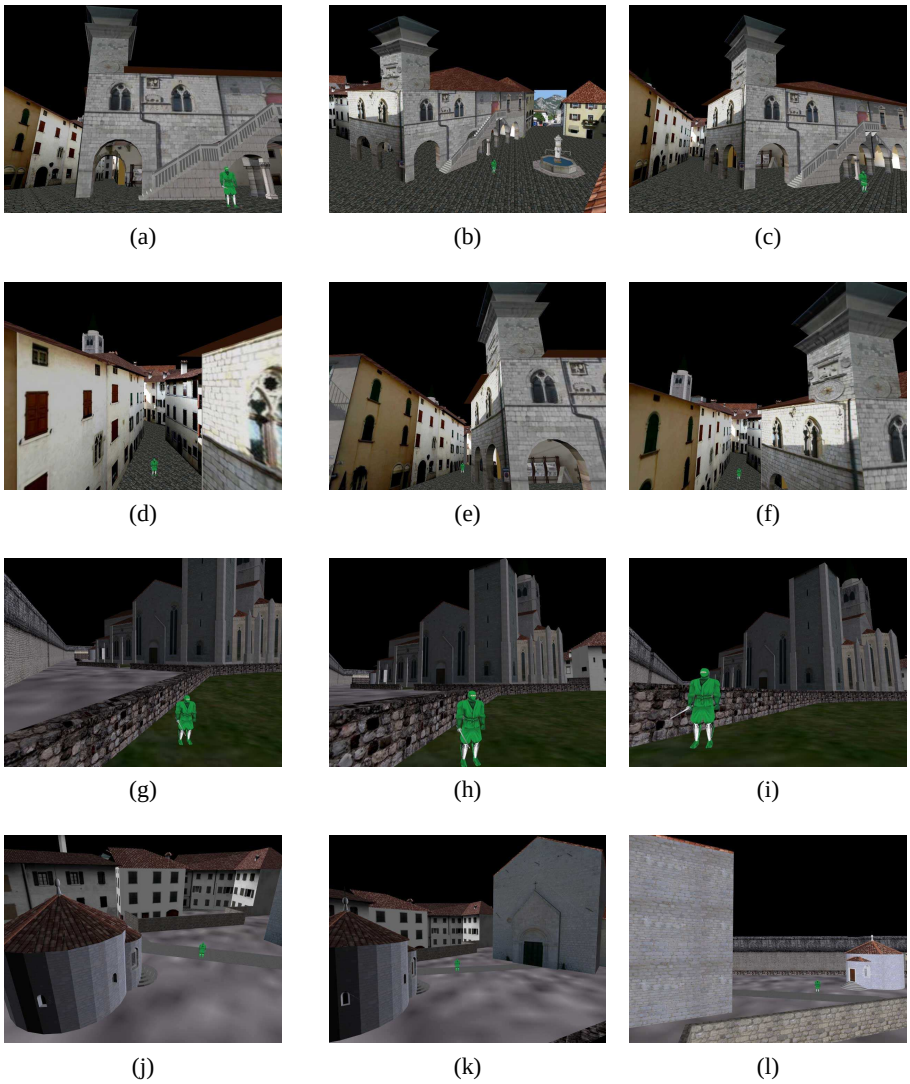
4.1 Performances

For each run, we have recorded the execution time (which includes both phases) and the quality of the generated camera, i.e. the best value of the fitness function. As it is shown in table 1, PSO is consistently faster than EXH, even when the grid resolution employed by EXH is ad-hoc set to minimize the number of considered points while still producing images with quality close to PSO. On the contrary, PSO parameters have not been changed across the problems. However, theoretically one could obtain better results by tuning the the size of the swarm depending on the number of properties defined. Note also that this result was not straightforward before doing the tests, since EXH works on a discretization of the search space, while PSO works in a continuous search space. With respect to the quality of the generated cameras, in the first two problems PSO obtains values of the fitness function close to the optimum (e.g. 4.87 out of 5 in the first problem), while in the last problem it is impossible to fully satisfy all requirements and therefore the degree of satisfaction is lower (28.73 out of 37.5).

As one can see from the table, one of the issues with PSO is the variance in the time needed. However, if there is the need of guaranteeing a solution within a certain amount of time, the algorithms has the nice property of being stoppable anytime and return a (possibly non optimal) solution.

	time (ms)					quality		
	PSO				EXH	PSO		EXH
Problem	min	max	average	std. dev.	average	best value	std. dev.	best value
over the shoulder (1.2)	33	319	151	36.24%	541	4.87/5.0	5.13%	4.84/5.0
occlusion (1.3)	58	737	443	36.95%	921	4.78/5.0	2.51%	4.80/5.0
navigation aid (1.4)	1762	3348	2235	16.63%	8993	28.73/37.5	6.41%	28.23/37.5

Table 1: Performance data collected in 100 runs for PSO and EXH



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

Fig. 3: Cameras computed from the properties listed in 1.4. Each row shows images computed with the user's avatar in different positions. For each row, the left image has been computed by EXH considering a $15 \times 15 \times 15$ grid of possible camera positions; the center and right images have been computed by PSO.

From the results obtained, another evidence is that execution times generally depend on the number and nature of the properties that are evaluated with fitness functions, with occlusions, as noted in the literature, being the most costly evaluation.

5 Conclusions

We have presented a PSO approach for the VCC problem. The proposed method works with static scenes and performs significantly better than an exhaustive search on a discretization of the space. Nevertheless, additional experiments on other scenes and image descriptions should be carried out to evaluate the method more thoroughly. With respect to this point, one goal for future work that could benefit the entire research area is to define a benchmark comprising a set of representative and realistic scenes and problems, thus allowing the experimental comparison of proposed methods in the literature.

Acknowledgments. The authors acknowledge the financial support of the Italian Ministry of Education, University and Research (MIUR) within the FIRB project number RBIN04M8S8.

References

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

[2] W. H. Bares, J. P. Gregoire, and J. C. Lester. Realtime constraint-based cinematography for complex interactive 3d worlds. In *AAAI/IAAI*, pages 1101–1106. 1998.

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

[4] M. Christie and P. Olivier. Automatic camera control in computer graphics. In *Proceedings of the Annual Eurographics conference - State of the Art Reports*, pages 89–113. 2006.

[5] R. C. Eberhart and J. Kennedy. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks 1995*, volume 4, pages 1942–1948. 1995.

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

[7] J. Kennedy, R. C. Eberhart, and Y. Shi. *Swarm Intelligence*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2001. ISBN 978-1-55860-595-4.

[8] J. H. Pickering. *Intelligent Camera Planning for Computer Graphics*. PhD thesis, Department of Computer Science, University of York, 2002.