

Visibility-Aware Framing for 3D Modelers

Roberto Ranon¹ and Marc Christie²

¹HCI Lab, Department of Math and Computer Science, University of Udine, Italy

²IRISA/INRIA, University of Rennes I, France

Abstract

Modelling and editing entire 3D scenes is a fairly complex task. The process generally comprises many individual operations such as selecting a target object, and iterating over changes in the view and changes of the object's properties such as location, shape, or material. To assist the stage of viewing the selected target, 3D modellers propose some automated framing techniques. Most have in common the ability to translate the camera so that the target is framed in the center of the viewport and has a given size on the screen. However, the visibility of the target is never taken into account, thereby leaving the task of selecting an unoccluded view to the user, a process that shows to be time-consuming in cluttered environments. In this paper, we propose to address this issue by first analyzing the requirements for an automated framing technique with a central focus on visibility. We then propose an automated framing technique that relies on particle swarm optimization, and implement it inside Unity 4 Editor. Early evaluations demonstrate the benefits of the technique over the corresponding standard Unity function, and trigger interesting perspectives in improving a simple yet mandatory feature of any 3D modelling tool.

1. Introduction

A repetitive task encountered by 3D content creators is the selection and framing of objects on which different manipulations are performed (such as translations, rotations, mesh editing, material editing). To ease the process and prepare the manipulation tasks, most 3D modeling packages (like 3DS Max, Maya) and 3D game editors (like Unreal Engine or Unity) offer a command to frame a selected object, i.e. move the camera so as to offer a centered view of the object.

Typically, the command is implemented by a viewport camera transformation composed purely by a translation, which (i) puts the selected object(s) (more precisely, its pivot) at the center of the current viewport, and (ii) sets the viewport camera at some distance from the object, so that it is entirely contained in the viewport.

The implementation of this functionality in most tools however does not account for the visibility of the selected targets. We have examined how two major modelling software, namely 3D Studio Max and Maya, and two major game/scene editors, namely Unity 4 and Unreal 4, implement the above described feature. With some minor differences, all programs implement the feature as described above, and suffer from this same issue. As a consequence, especially in complex scenes, it is common that the selected

object(s) are not visible at all. The computed view is therefore unadapted to perform the intended manipulations, and the user needs to manually move the camera through classical viewpoint manipulators (translation, rotation or compositions such as arcball) until a suitable view is reached. This leads to losing some time and effort. While there are straightforward ways of handling occlusion, such as hiding the geometry of the occluders, or using wireframe representations, these generally prevent the user from performing the intended manipulation tasks on the selected object (e.g., precisely placing an object on top of another one, changing the color/material to mimic its environment).

While one could state that each of these manual changes on the camera do not require a significant time or effort, these tasks are repeated many times. The benefit of replacing this default feature by an automated visibility-aware viewpoint computation technique is therefore appealing, as also noted by Phillips et al. [PBG92], who proposed an hemicube approach to derive regions around an object that guarantee visibility and where the viewport camera can be moved. However, their approach works only for single objects, and more importantly does not work in closed scenes, i.e. those situations where occlusions are more problematic.

We propose an alternative strategy, based on automat-

ically computing a view that, besides respecting the two above goals, takes also into account occlusion, and then perform a combined translation and rotation of the viewport camera to frame the selected object (or even multiple selected objects) such that primarily occlusion, and then camera rotation, are minimized. The process is based on an efficient viewpoint optimization process that relies on particle swarm optimization [EK95]. Our approach can also handle multiple objects, and is not limited by the spatial configuration of the scene. We describe how we have implemented the feature in Unity, and experimentally demonstrate its benefits over the corresponding Unity-provided framing function.

While an extensive experimental evaluation with users has not been performed yet, informal feedback from regular Unity users has been generally positive, and has also provided interesting directions for future work.

The paper is structured as follows. In Section 2, we describe in detail the motivations and state the requirements. In Section 3 we present our approach, and display different examples. In Section 4 we experimentally compare our approach with the corresponding standard Unity functionality. Finally, in Section 5, we discuss our results, and outline directions for future work.

2. Motivations

In this Section, we motivate the need for our approach by using an example scene, shown in Figure 1a, to demonstrate how the framing command works. The viewport camera is initially positioned and oriented as shown in Figure 1b, where the bottom image is the view from the viewport camera. The top image in Figure 1b shows also two objects that we will consider: a computer monitor, which is positioned to the right of the camera and in the same room, and a plant, which is positioned in an adjacent room.

In Unity 4 Editor, after selecting one or more objects, either in one of the visible viewports or by choosing it from a list of objects, a user can invoke the *frame selected* command to frame the selection. In Figure 2 we show the effect of the command after selecting, respectively, the computer monitor or the plant. In the case of the computer monitor (see the two top images in Figure 2), the result is a viewport where the selected object is fully visible. In the case of the plant, (see the two bottom images in Figure 2), the result is a viewport where the selected object is not visible at all, since there is a wall between it and the viewport camera. Since the plant is right behind the wall, there is no way to make it visible by purely translating the viewport camera. In this last case, only the plan transform gizmo (i.e. the axes in the figure) provide the user with a hint that the object is somewhere behind the wall. In Unity, the transition between the old and new viewport cameras is implemented through a smooth animation: in this way, the user can see how he is moving through the scene, and maintain spatial coherence and thus orientation.

We reviewed this automated framing technique for tools such as 3DS Max Studio, Maya and Unreal game engine. All follow the same idea, with the minor differences that are presented in Table 1.

From this initial example, as well as Table 1 we can draw a number of requirements for a better framing technique:

1. **visibility:** it represents the central requirement to improve any automated framing technique. Computing the visibility of a target in a 3D modeler can be done in a cost-effective way by using ray-casting, which is usually accessible to scripts or plug-ins. However selecting a viewpoint that ensures a given visibility shows to be far more complex. There are several approaches in the literature, with [CON08] discussing the problem in detail and presenting the state of the art up to year 2007. More recent approaches can be broadly divided into two categories: those that search for camera path that ensure the visibility of targets (e.g., [CNO12, OSTG09]), and those that aim at computing a viewpoint anywhere that ensure that one or multiple targets are visible (e.g., [RU14, BY10]).
2. **size:** given that the selection stage precedes a manipulation stage, it appears essential to display the selected object in relation to its environment, therefore at a given size. For example, standard deviation for size of selected targets, with Unity's framing tool is 6% of the screen size $+[-2, 2]$, which provides a clear enough view on the target and ensures establishing the object in relation with its background. The on-screen size is, in an obvious way, related to the task to be performed. In Unity, most tasks are related to displacements, rotations or property editing (in contrast with mesh-editing). General purpose modelling tools provide larger views on selected targets. In fine, the key is to provide means for an automated tool to adapt size to the task.
3. **spatial cognition:** establishing the spatial relation between the previous location, and the one close to the selected target is of prime importance. While linear camera motions proposed in most tools maintain this relation (given the simplicity of the path), more evolved movements that would require moving around the target to provide an unoccluded view may fail to establish or preserve the understanding of this spatial relation. Therefore, if such rotations are necessary, means should be provided to minimize the change in camera angle.

3. Our approach

We propose to substitute the framing mechanism described in the previous Section by a more sophisticated approach, called *visibility-aware framing*, which aims at computing a new viewport camera that:

- (a) visualises the selected object the center of the viewport, but is allowed to change position *and* rotation with respect to the starting viewport camera;



(a) Our example scene

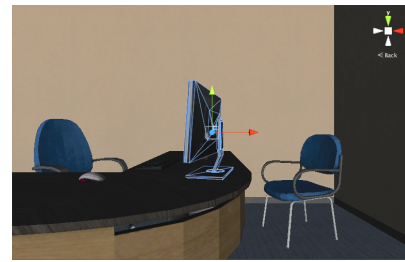


(b) Starting camera, and example objects

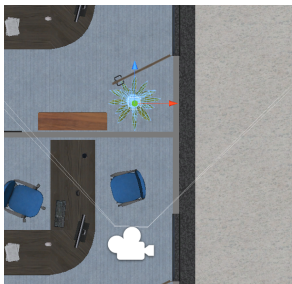
Figure 1: Our example scene represents a building floor with a corridor and nine offices. Offices contain furniture, such as chairs, computers, desks. On the top right image, we see the initial viewport camera position and orientation, and the two objects we are considering (depicted in cyan, with local coordinate axes): a computer monitor, and a plant. On the bottom right figure, we see the starting viewport view from the camera. Screenshots are taken from the Unity Editor.



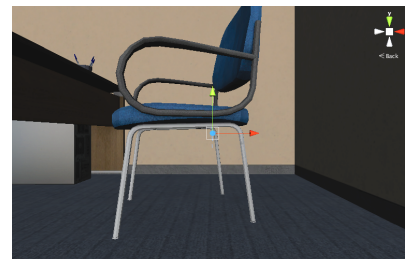
(a) Viewport camera position and orientation after framing the monitor



(b) Viewport after framing the monitor



(c) Viewport camera position and orientation after framing the plant



(d) Viewport after framing the plant

Figure 2: Results of using Unity 4 *Frame selected* command on the monitor (images a and b) and on the plant (images c and d), each starting from the initial camera position illustrated in Figure 1a

Software	Command	Behaviour
3D Studio Max	zoom extents selected (zoom extents all)	translates the viewport camera (all viewport cameras) such that the selected object (or the selected objects) is framed at the center and entirely contained in the viewport, without taking into account visibility. The transition between the starting and computed camera is immediate.
Maya	Frame Selection	translates the viewport camera such that the selected object (or the selected objects) is framed at the center and entirely contained in the viewport, without taking into account visibility. The transition between the starting and computed camera is immediate.
Unity 4 Editor	Frame Selected	translates the viewport camera such that the selected object (or the selected objects) is framed at the center and entirely contained in the viewport, without taking into account visibility. The transition between the starting and computed camera is performed through an animation.
Unreal 4 Editor	Focus selected	translates the viewport camera such that the selected object (or the selected objects) is framed at the center and contained in the viewport at least in one dimension, without taking into account visibility. However, the contour of the object is rendered on top of the scene, so that even with occlusion the viewer can limitedly understand where the object is and its shape. The transition between the starting and computed camera is performed through an animation.

Table 1: Automatic object framing functionalities in popular 3D modellers and editors.

- (b) visualises the selected object *with less possible occlusion by other objects in the scene*;
- (c) visualises the selected object entirely contained in the viewport, and occupying an area around 6% of the viewport;
- (d) minimises rotation with respect to the current viewport camera.

In the case of multiple selected objects, the midpoint of all their bounding box centers will be centered in the viewport, and the required area of each projected object will decrease according to how many objects are selected, while the other requirements stay the same. Hereinafter, we will, for simplicity, consider the case of one single selected object.

The possibility of changing rotation with respect to the starting viewport camera is fundamental to guarantee visibility, as shown in the example in the previous Section. The second requirement introduces visibility enforcement. The third requirement constrains the size of the projected object: the 6% value was derived to mimic Unity behaviour. In this way, the user will see some context around the object, understand its location and possibly perform limited translations / rotations / scalings without having to change the viewport camera. The reason for the last requirement, i.e. to minimise camera rotations when transitioning from the current viewport camera to the new one, is to reduce the time needed for a user to regain spatial orientation and awareness in the scene after the viewport camera transition.

To compute the viewport camera that frames an object in the scene as specified above, we employ the Viewpoint Computation library described in [RU14]. This approach is able to compute, in a given amount of time, the virtual camera that best satisfies a list of visual properties. The visual properties can express desired values of the size (area, width or height), visibility, camera angle and on-screen position for any choice of objects in a 3D scene. From an input list of visual properties, the library first builds a function that returns a numeric value indicating to what extent a given virtual camera satisfies the properties. Then, a solver based

on Particle Swarm Optimisation [EK95] iteratively searches the 3D scene for the virtual camera that maximises the satisfaction function. In principle, the library works with any type of scene or object and does not require any preprocessing of the scene, and relies on the rendering engine to obtain information about the position and size of the bounding volumes of objects and to perform visibility queries, e.g. through ray casting. Most, if not all 3D modelling packages and editors, provide these information through a plugin API or internal scripting language. We refer the reader to [RU14] for a detailed explanation of the viewpoint computation approach. C++ source code of the library is available at <http://bit.ly/1wdBOqq>. In the following, we will describe in more detail the features of the library that are relevant for this paper.

In the library, a virtual camera is defined by 8 real-valued components: three coordinates for the position (pos_x , pos_y , pos_z), three coordinates for the look-at point ($look_x$, $look_y$, $look_z$), a *roll* angle to define the horizon, and a FOV parameter. The available visual properties are defined in Table 3. The satisfaction of each property can be defined by a user-provided linear spline with an arbitrary number of points, where the x values are in the domain of the specific property, and y values, i.e., satisfaction, are defined in the $[0,1]$ interval. The satisfaction of a virtual camera is then defined as a weighted sum of each individual property satisfaction function, where weights are user-provided real numbers that encode a property importance with respect to the others. The search space used by the solver can be defined by an axis-aligned bounding box for the camera position, another one for the look-at point, and two intervals of real values, one for the camera roll, and one for the camera FOV.

Going back to our problem, requirement (a) is implemented by setting the search space for the look-at-point to the center of the selected object, and the search space for the camera position, to a bounding box which is centered on the selected object, and whose size is double the size of the scene. Additionally, we block the roll angle to zero to avoid

Property	args	Semantics
Size	t, D	<i>Area, Width, or Height</i> (the possible values of D) of t_v in viewport-relative coordinates
Framing	$t, Rect$	the fraction of t_v which is inside $Rect$
RelativePosition	s, t, RL	the fraction of s_v which is <i>right, left, above, or below</i> (the possible values of RL) any point of t_v
Occlusion	s, t	the fraction of s_v which is occluded by t_v . t can be equal to <i>scene</i> , which means every object except s , when we want to take into account any source of occlusion
Angle	t, \mathbf{u}	the angle between vector \mathbf{u} and the vector from t to the viewpoint; \mathbf{u} can be also defined by the keywords <i>front, up</i> which respectively are t front and up vectors

Table 2: Available properties in the viewpoint computation approach we adopt. In the table, s, t are *targets*, i.e., objects in the scene; $Rect$ is a 2D rectangle in viewport coordinates. In the third column, t_v indicates the projection of the bounding box of target t from the viewpoint v , with the exception of the *Occlusion* property, for which t_v is defined by an arbitrary number of rays from v to points in t bounding box.

dutch angles, and we fix the FOV to the current editor camera FOV setting.

Requirement (b) is implemented by an *Occlusion* property, whose satisfaction function is illustrated in Table 3. Full visibility (i.e. zero occlusion) will give full satisfaction; until half visibility, there is a slight decrease in satisfaction (0.8), and then less than half satisfaction will entail a satisfaction which is close to zero.

Requirement (c) is implemented by a *Size* property, whose satisfaction function is illustrated in Table 3. Until projected size reaches a minimum value (3%), satisfaction is close zero; it linearly increases from 0.8 (3% size) to 1 (6% size) and then decreases gradually until 0.1 (50% size). A greater size will entail a satisfaction which is close to zero.

Requirement (d) cannot be implemented with the properties available in the library. We therefore added a new property, called *CamOrientation*, that computes the angular difference between a provided orientation (which will be the orientation of the starting viewport camera) and a candidate camera orientation. Its satisfaction function is illustrated in Table 3. A zero difference will give maximum satisfaction, and the decrease is linear until zero satisfaction is reached with maximum angular difference.

Given these properties, our viewpoint computation problem becomes:

$$\begin{aligned} & \min(1.5 * sat_{Size}(Size(t, Area)) + \\ & 2 * sat_{Occlusion}(Occlusion(t, scene)) + \\ & sat_{CamOrientation}(CamOrientation(\mathbf{r})) \end{aligned} \quad (1)$$

Where t is the selected object, and \mathbf{r} is the rotation of the starting viewport camera. In the next Section, we describe how we have implemented the visibility-aware framing command in Unity.

3.1. Implementation

We have extended the Unity editor such that, besides the standard *frame selected* functionality, one can use our modified version by invoking a menu command or a shortcut key.

The extension is implemented as a C# script that can access various editor variables and states (e.g., the currently selected object in the viewport, the viewport camera), as well as the scene (e.g. bounding boxes, performing ray casts). Finally, the script implements also the library described in [RU14], with the additions mentioned above.

To measure visibility, we use Unity-provided ray intersection queries. More specifically, we cast six rays from a candidate camera to the object bounding box center and other 5 random locations inside the bounding box. Visibility is then computed as the ratio of rays that do not intersect any other object collider. In our implementation, we use full detail meshes as colliders to maximise precision in computing visibility. Of course, there could be more precise methods of computing visibility (from ray casting to an object vertices, instead of random points inside the object bounding box, to occlusion queries). However, the chosen method is fast and, and has given good results in our experience, as we will show in Section 4. Since ray intersection queries in Unity are computed through the Physx library, this approach has the little inconvenience that any object in the scene must be also represented in Physx, i.e. have an associated *collider* mesh, which needs to be previously added in the editor, as this is the only method that Unity allows. By having access to Unity source code, however, this issue could be likely eliminated.

To measure projected size, we use the object axis-aligned bounding box, and measure its projected area using the approach described in [ST99]. Our understanding is that Unity uses, for its framing function, the object bounding sphere. The two methods give similar results for objects that have roughly the same size in the three dimensions, but will give different results otherwise. We think the bounding box is better, as, for objects that are much longer along one side, the bounding sphere method will result in a much smaller size on the viewport. However, this could be also a matter of personal user preferences. We could use oriented bounding boxes to get a more precise measurement of size, but in this application we are not concerned with precise composition, so it would not be probably worth the extra effort. More precise methods, such as off-screen rendering, besides being too complex to compute for viewpoint computation

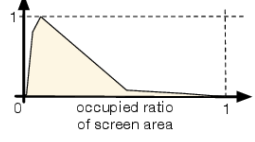
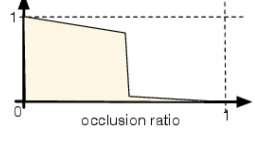
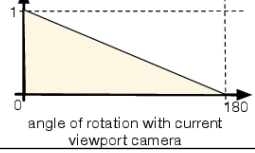
Property Type	Semantics	Weight	Satisfaction Function
Size	the target object should cover around 6% of the screen area	1.5	
Occlusion	the target should not be occluded by other objects	2.0	
CamOrientation	angular difference with current viewport camera	1.0	

Table 3: Properties defined in our approach to compute a virtual camera.

purposes [RU14], would turn out to be problematic for some kind of objects, e.g. objects with many holes, which would be framed at more distance than other objects occupying a comparable estate on screen.

We set the time allowed for viewport camera computation to 30 milliseconds, so that there is no perceived delay when calling our framing function, and use exactly the same standard Unity animations to transition from the actual viewport camera to the newly computed one.

Figure 3 shows how our approach handles the example in Section 2. In both cases, our approach manages to compute camera viewports that make the selected object fully or almost fully (in the case of the plant) visible. In the case of the monitor, since no change in orientation is required to ensure visibility, the computed camera is very similar to Unity standard framing computed one (see Figure 3a).

4. Experimental Results

To compare our approach with the standard Unity *frame selected* command, we have setup an experiment in which we select a random object from a random viewport camera position and orientation, invoke the standard Unity and the visibility-aware version of the *frame selected* command, and compare the results. The experiment uses the building scene in Figure 1a. Our comparison is based on measuring selected object visibility and amount of rotation with respect to the starting camera. What we expect is that visibility-aware framing command should obtain much better results in terms of visibility, while not introducing much rotation with respect to the starting camera. As mentioned in the previous Section, the time allowed to our approach to perform its computation is 30 ms.

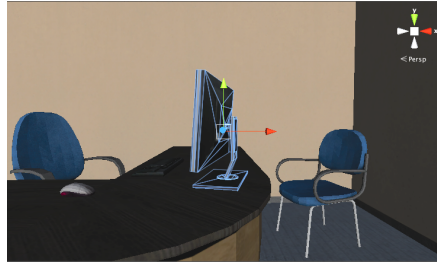
We also measure object projected size and overall satisfaction of the result (with respect to the properties defined in the previous Section) for both the approaches. What we expect is that our visibility-aware framing command should obtain similar results in terms of projected size, as it is designed to do so. Overall satisfaction should also be higher in the case of our approach, since the standard Unity framing command will be penalized for not taking into account visibility.

To measure visibility, we use here a more precise approach than the one in our framing function. More specifically, we shoot 30 rays, each from the camera position to a random vertex of the object, and define visibility as the ratio of rays that do not intersect any other object (therefore, 0 equals to full occlusion, 1 to full visibility). For the size of the selected object, we keep the axis-aligned bounding box method explained before.

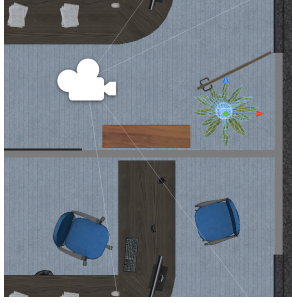
Results on the example scene described in Section 2 for 1000 comparisons are presented in Figure 4. More specifically, Figure 4a shows the distribution of visibility across all the comparisons. In the case of our approach, all measured values, with the exception of some isolated cases (the dots in the plot) are above 0.6 (i.e. at least 560% of the object is visible), the second quartile is around 0.8, and the median is above 0.9 (i.e. the object is almost fully visible). On the other hand, for the standard Unity framing function the resulting visibility is much more scattered across the whole range, with a median value of 0.66. Moreover, among the 1000 tests, complete occlusion (i.e. visibility = 0) was found in 345 cases with the Unity framing command. Finally, with respect to visibility, the Wilcoxon-Mann-Whitney test confirms that our framing function is significantly better than Unity one (with $p < 2.2e-16$).



(a) Viewport camera after framing the monitor



(b) Viewport after framing the monitor



(c) Viewport camera after framing the plant



(d) Viewport after framing the plant

Figure 3: Results of using our framing command on the monitor (images a and b) and on the plant (images c and d), each starting from the initial camera position illustrated in Figure 1b

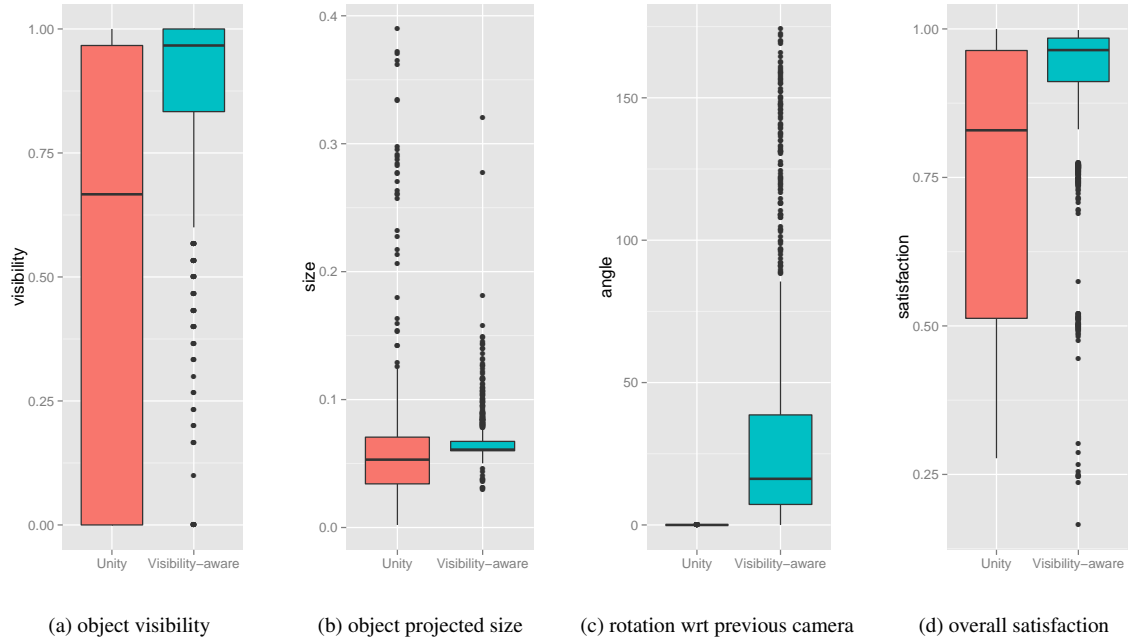


Figure 4: Box plots showing the distribution of visibility, size, rotation with respect to previous camera, and satisfaction in the framing viewport cameras computed by Unity Editor and our visibility-aware framing command. Dots represent outliers. Data collected over 1000 comparisons in the building scene in Figure 1a

Figure 4c shows the distribution of the angle of rotation of each found camera with respect to the starting one. For Unity, all measured values are obviously zero, as there is no rotation; for our approach, the median value is around 16 degrees, with only isolated cases exceeding 90 degrees.

Figure 4b shows the distribution of projected size across all the comparisons. Our approach behaves quite similarly to Unity standard framing function, with less variability in results. This could be due to the fact that Unity is using the bounding sphere of the target to compute the camera distance from it. At practical level, however, our projected size is very similar to Unity one.

Figure 4c shows the distribution of satisfaction across all the comparisons. For our framing approach, it shows that in the majority of cases we are able to reach a satisfaction above 90%, with the first quartile starting at around 80%. The plot also shows that, in some isolate cases, our approach finds viewpoints with quite low satisfaction. This is not due to particularly difficult situations, but to the fact that, sometimes, PSO initialisation is particularly unfortunate, and the solver is unable to derive a good solution in the allowed time. In those cases, we could simply restart search before presenting the result to the user, and very likely obtain a good result, at the price of a little lag in the interaction with the editor.

5. Discussion and Conclusions

Our experimentation, albeit limited to one scene, shows that our visibility-aware framing function is able to effectively substitute the Unity standard function, and in addition provide at least partial, and in the vast majority of cases good, visibility of the selected object. Moreover, rotation with respect to the starting viewport camera is generally quite limited.

Of course, measuring the improvements in visibility of our approach over the standard Unity framing functionality does not tell us whether users would prefer it. While a formal experiment with users has not been carried out yet, preliminary reports from a few Unity expert users that tried our framing functionality were generally positive. Users appreciated the increased visibility, and stated that the new function would have saved them a few viewport camera manipulations. Our next step is to compare our feature and the Unity one in a controlled experiment with users. In particular, we are interested in understanding whether users like our visibility-aware framing function, and whether it also translates into effective savings of time in performing a sequence of object manipulation tasks.

In the early informal sessions with users, some of them complained that, in case of relevant changes in viewport camera orientation, they initially felt disoriented and had to take some time to understand the new view on the scene. This could be also due to the fact that Unity built-in viewport camera animations are very short in time, and therefore

considerable rotations coupled with translation result in very quick changes in the viewport display, making it difficult for a viewer to follow. Possible solutions could be to make camera rotations slower in the animation, or even perform translations (possibly with very limited rotation) first until the object is framed at the center, and then rotate around it until visibility is reached.

Another interesting aspect that came up with users was about their preferences among possible viewpoints that guarantee visibility. For example, some users stated that they typically like to view objects to be manipulated with an angle of 45 degrees from the up vector (i.e. between a top and front view) without degenerate local axes [PBG92], and the visibility-aware framing function could be modified to take this into account. More generally, the best view could depend on user's preferences, as well the kind of task (e.g. translation versus material manipulation). To take all this into account, one could also generate multiple alternative views, e.g. to be displayed in small viewports, so that the user can choose the one that best suits her preferences or task.

6. Acknowledgments

Roberto Ranon acknowledges the partial support of the PRIN 2010 project 2010BMCKBS_013.

References

- [BY10] BURELLI P., YANNAKAKIS G. N.: Global Search for Occlusion Minimisation in Virtual Camera Control. In *IEEE Congress on Evolutionary Computation* (Barcelona, 2010), IEEE. 2
- [CNO12] CHRISTIE M., NORMAND J., OLIVIER P.: Occlusion-free Camera Control for Multiple Targets. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2012), Eurographics Association, pp. 59–64. 2
- [CON08] CHRISTIE M., OLIVIER P., NORMAND J.-M.: Camera Control in Computer Graphics. *Computer Graphics Forum* 27, 8 (Dec. 2008), 2197–2218. 2
- [EK95] EBERHART R. C., KENNEDY J.: Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks 1995* (1995), vol. 4, pp. 1942–1948. 2, 4
- [OSTG09] OSKAM T., SUMNER R. W., THUREY N., GROSS M.: Visibility transition planning for dynamic camera control. In *2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation - SCA '09* (New York, New York, USA, 2009), vol. 1, ACM Press, pp. 55–65. 2
- [PBG92] PHILLIPS C. B., BADLER N. I., GRANIERI J.: Automatic viewing control for 3D direct manipulation. In *I3D '92: Proceedings of the 1992 symposium on Interactive 3D graphics* (Cambridge, Massachusetts, USA, 1992), ACM Press, pp. 71–74. 1, 8
- [RU14] RANON R., URLI T.: Improving the efficiency of viewpoint composition. *IEEE Transactions on Visualization and Computer Graphics* 20, 5 (2014), 795–807. 2, 4, 5, 6
- [ST99] SCHMALSTIEG D., TOBLER R. F.: *Real-time Bounding Box Area Computation*. Tech. Rep. TR-186-2-99-05, Jan. 1999. 5