SHOCam: A 3D Orbiting Algorithm

Michael Ortega[1], Wolfgang Stuerzlinger[2], Doug Scheurich[3]
[1]Laboratoire d' Informatique de Grenoble, UMR 5217 F-38041, Grenoble, France,
[2]Simon Fraser University, Vancouver, Canada, [3]York University, Toronto, Canada

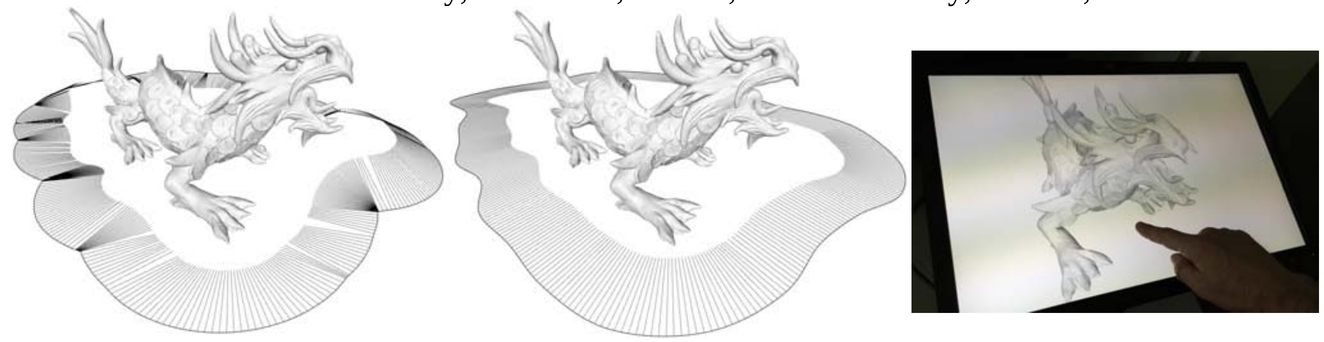

Figure 1. Orbit comparison between HoverCam (left) and SHOCam (middle). Discontinuities in HoverCam's camera *motion* are visible in corners where the camera motion stops. Discontinuities in HoverCam's camera *direction* are visible along the path through the irregular spacing of the view vectors. SHOCam generates substantially smoother paths and view direction changes. The right image shows interaction with SHOCam on a touchscreen.

ABSTRACT

In this paper we describe a new orbiting algorithm, called SHOCam, which enables simple, safe and visually attractive control of a camera moving around 3D objects. Compared with existing methods, SHOCam provides a more consistent mapping between the user's interaction and the path of the camera by substantially reducing variability in both camera motion and look direction. Also, we present a new orbiting method that prevents the camera from penetrating object(s), making the visual feedback – and with it the user experience – more pleasing and also less error prone. Finally, we present new solutions for orbiting around multiple objects and multi-scale environments.

Author Keywords
3D user interfaces; 3D navigation; 3D orbiting

ACM Classification Keywords
H.5.2. Information interfaces and presentation (*e.g.*, HCI): Interaction styles.

INTRODUCTION

Thanks to recent technological advances, 3D graphics is ubiquitous today. Based on this capability, interactive 3D environments are now accessible on phones, tablets, desktop computers, and even on public displays. 3D user interfaces are concerned with the interaction with such virtual 3D graphics environments. One of the most frequent actions is viewing of 3D models, *e.g.*, before 3D printing an object. To support this kind of interaction, it is critical to provide intuitive, easy to learn, and efficient 3D navigation techniques for novices as well as for experts. Recent work has introduced advanced methods to facilitate **orbiting**, *i.e.*, **moving around** an object while **facing it** from a **constant distance**. Yet, the proposed methods still have limitations and can lead to situations that can confuse the user or yield undesirable behaviors, such as jittery navigation paths.

3D orbiting can be considered as an "observation task". Observing an object involves (1) navigation around it for observing said object or its parts from different angles, and (2) moving closer to or further from it for seeing details or getting an overview. This can be achieved with only 3 degrees of freedom (3 DOF) control [10]: two DOF's for moving "along" the surface of the object (a combination of rotation and translation around the object), and a single DOF for moving closer/further away from the object. This kind of navigation method, which automatically combines rotation and translation around the object, is something that even novices easily understand, as evidenced by the many 3D viewing systems on the web that use these conventions.

3D orbiting can also be seen as a "navigation task". Here a new constraint appears: the whole scene affects the camera behavior, as the system ideally should never permit the user to move the camera into objects. The (unexpected) view of an interior of an object can be confusing, especially for novices. This is well known from computer games. In such situations, novices may get "lost" and then have to recover a reasonable camera pose with trial and error. Here, the

© 2015 ACM. ISBN 978-1-4503-3779-3/15/11…$15.00
DOI: http://dx.doi.org/10.1145/2807442.2807496

above-mentioned 3 DOF interaction method is still a good solution, but the algorithm that determines the camera path and velocity should take the rest of the scene into account.

Existing methods do not address all of these constraints. We first review previous work and then present our new 3D orbiting methods. In the discussion section we illustrate our method with several examples and finish with future work.

Previous Work

There is a large body of literature for 3D navigation. Camera control methods [4] are also related. We focus here only on orbiting methods and automatic speed control. Early work on orbiting, such as Unicam [23], focused on orbiting around points or the center of an object. Safe 3D Navigation [7] highlighted that off-screen orbit points cause many forms of navigation problems. Moreover, orbit points that are not on, or close to, the surfaces of the scene are also problematic. Smooth view transitions were already identified to be desirable [18]. Recent work confirmed that smoother camera motions reduce cybersickness [6]. Phillips [18] introduced a method that avoids obstructing the view of the object during orbiting. Niewenhuisen [15] presented an automatic approach to navigate a camera between objects with collision avoidance, but only for 2D scenarios.

The HoverCam orbiting method [10] keeps the viewer at a constant distance relative to the *surface* of the object. McCrae *et al.* [13] improved HoverCam's efficiency in their multi-scale navigation approach. They also identified that HoverCam may switch to other close-by objects during orbiting, but then simply ignored other objects while orbiting or stopped the viewer when they would orbit into another object. Trindade *et al.* emphasized that the user needs to stay in front of a surface while orbiting [21].

Recently, touch-based 3D navigation has become popular. Navidget [8] uses overlaid widgets with different zones to activate different navigation behaviors, such as orbiting, which leads to a constant switch in attention between the zones and scene. Drag'n Go [14] implements steering towards a point of interest. Finally, ScrutiCam [5] presented a touch-based orbiting method, which does not handle objects with concavities very well.

Some 3D sketching systems ensure that the system controls the view position to facilitate drawing on curved surfaces [2], which is similar to orbiting. A new method has extended this to drawing on 3D surfaces [16], to ensure that the viewer can always see the surface straight on and access the whole surface of an object in a single drawing gesture.

Mackinlay [12] first observed that setting the camera speed proportional to the current distance to a target point works well. Ware confirmed this [22]. In speed-coupled flying with orbiting [20] and whenever the viewer is flying fast, the viewer eye-level is automatically elevated to give the user a better overview of the environment. In a comparison of multi-scale steering methods, Kopper [11] identified that automatic size or speed control worked far better than manual size or speed control in multi-scale scenes. Zhang [21] confirmed that linking size and speed in a multi-scale traveling approach is beneficial. McCrae *et al.* [13] computed a vector to push the viewer away from nearby geometry. Trindade [21] identified that speed control via the global minimum distance can slow the user down unnecessarily. They detect situations where the viewer can speed up with a ray in the view direction. Argelaguet found no strong difference between distance-based speed control and a method that keeps optical flow constant [1].

Motivation

The overarching objective of our work is to generate smooth camera movements for 3D navigation. Several existing navigation methods, such as look-around, already generate smooth camera motions. Look-around is universally mapped directly to 2 DOF control. Another common design decision is that the viewer is prevented from "flipping over", *i.e.*, looking at the scene upside down, as people easily become confused by this.

In 3D modeling and computer games, panning/strafing moves the viewer orthogonal to the view direction in the horizontal plane. In orbiting/circle-strafing the camera moves around a specific object. Our work targets such orbiting around objects. We believe that the fundamental idea of having the camera stay at a constant distance to the surface is appropriate for this kind of navigation task. Yet, we also believe that said distance can vary to a limited degree, if this leads to better camera motions. One common problem with both panning and orbiting is that the user can easily pan or orbit into other objects if the system does not prevent the camera from doing this. With camera collision detection, the camera motion may be stopped or altered abruptly upon such a collision. This issue is exacerbated by the fact that most graphics applications use a FOV of less than 90 degrees. This prevents humans from using their peripheral vision to prevent collisions.

A common problem for driving or flying methods with manual speed control is that users overshoot their targets. Then, users must stop, reverse and then move forward again, or stop, turn around, move forward at a reduced speed, and turn again towards their original destination. Following previous work [1,11,21,22,21], we use automatic speed control, even during orbiting. Another issue with manual speed control is that users can fly into objects. This is especially tricky when reversing, because most 3D systems do not enable users to see behind themselves, *i.e.*, provide no rearview mirror. This makes it hard to predict when the user will back into an object. Worse yet, backing into an object yields visually surprising results and causes confusion. Automatic speed control avoids all these issues.

Finally, we also want to target orbiting for scenes with whole groups of objects. One shortcoming of existing methods is that the orbit center always stays fixed when the user moves further away or comes closer. Yet, if the object becomes (too) small or too large, this heuristic fails.

Contributions

The main contributions presented here are:

- a method for smoother interactive orbiting;
- a new way to avoid collisions with surrounding objects during orbiting;
- a new method for multi-scale orbiting, *i.e.*, orbiting of object groups.

A NEW METHOD FOR 3D ORBITING - SHOCAM

With previous orbiting methods, such as HoverCam, the camera always stays at a constant distance to the surface and faces the closest point of the object surface. This simple principle is not quite powerful enough and different methods were proposed for avoiding sudden changes of the camera direction [10,13]. There are three main issues that have to be solved in this kind of orbiting approach: "how to turn around a corner", "how to turn to the next closest surface", and "how to guarantee smooth camera motions". We now discuss each one of them. As we investigate orbiting for complex scenes, such as multi-scale environments, and dynamic geometry, such as CAD editing or animation, we do not consider pre-computation [3,9,17].

Turning around corners: In the HoverCam approach, the optical flow changes on convex corners (see Figure 2a). When the camera reaches a "corner" of a convex surface, *i.e.*, an edge between two non-coplanar polygons of the mesh or even a vertex, the closest point of the surface will temporarily be "glued" to the feature until the camera faces the new polygon. As the camera still moves while the viewpoint stays on the edge, the optical flow changes abruptly, making this a very noticeable discontinuity in the visual feedback. This is critical for curved surfaces, as such surfaces are traditionally approximated and represented as a mesh of small polygons. Orbiting around such a surface then generates frequent start and stop camera motions. At some distance this can even look like camera tremor.

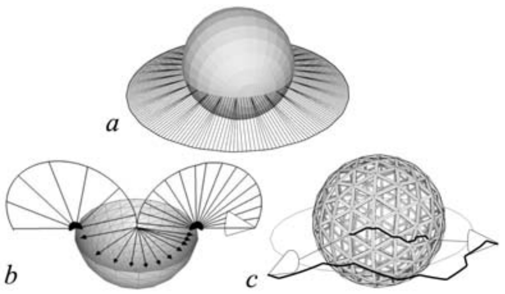

Figure 2. Issues with existing orbiting methods: *a* – viewpoint stops on edges of convex objects; *b* – camera stops on concavities (figure from [10]); *c* – both viewpoint and position of the camera may move up and down on complex surfaces.

In HoverCam, the authors proposed to address this "shakiness" issue by smoothing the model mesh or its normals. Yet, the degree of tessellation can affect only a limited range of camera distances. For smoothing normals, the authors propose to enable this only when the camera is further away from the object, in order to keep the ability of traveling along a flat polygon when the camera is close to it. However, this implies that the mesh tessellation has to be globally consistent in terms of the size of the output polygons, which is non-trivial to maintain in all situations.

In an improved HoverCam version [13], the authors proposed an image-based technique to approximate a local smoothing of the geometry. The authors specifically mentioned a box-filter. Yet, such smoothing does not work well around concavities or holes due to steep depth discontinuities in such areas. Indeed, when the camera travels forward, any geometry that comes into view at the sides of the view may suddenly become the closest point. Image-based smoothing cannot address this problem.

Turning to the next closest surface: The camera motion generated by HoverCam stops abruptly on concavities or surface holes (see Figure 2b). Here, HoverCam stops the camera motion completely and only interpolates the viewpoint direction. Camera movement resumes only when the final direction is reached. This leads to abrupt changes in the camera behavior. Sometimes a jump in the view direction also implies a discontinuous jump in the camera position. Then, HoverCam slows the camera down for interpolating the camera position. Overall this leads to piecewise constant optical flow, with discontinuities in between. Informally speaking, stopping or slowing down the camera motion in this way can cause the impression that the camera is "sticky", which may perturb the user. In the presence of several concavities, the camera motion is not fluid, and the start-stop nature of the motion can become annoying. Normally the user manipulates the camera in two dimensions (up/down and left/right) during orbiting. Yet, in concavities, the user loses control of one dimension, as user input is not mapped to movements anymore, but to the interpolation parameter to turn to the next closest point. This leads to a mismatch between user input and the viewpoint motion, especially when the direction of the interpolation is orthogonal to the initial user movements.

Smooth camera motion: Existing methods always look at the closest point on the object and thus both the look direction and the camera position correspond to the closest distance to the object. With this and during horizontal orbiting, both camera direction and position can still move vertically on geometrically complex surfaces (see Figure 2c) and vice versa. Yet, in the presence of concavities or holes in the object, the closest point will never stay at the same height, causing rapid and potentially large changes in camera direction. This again leads to tremors in the visual flow and to undesirable orbiting behaviors. Existing methods do not mention adequate solutions for this "shakiness" issue. Smoothing the normals or the geometry does not solve this issue, as both methods do not take the distance and relative pose of the camera into account.

All of the methods proposed by HoverCam and its evolutions smooth the changes in camera direction only for a subset of all potential situations. Moreover, they do not prevent large variations in camera motions. Depending on the particular geometry of the object, the camera thus can unnecessarily slow down and speed up, giving the user inconsistent visual feedback. This constantly varying mapping between the visual flow and the user's movements makes it more difficult for the user to navigate around an object in an easily predictable and controllable manner.

The following sections describe our new *SHOCam (Smooth HoverCam Orbiting Camera)* method, which guarantees smooth and constant camera motions, as well as smooth changes in camera direction for any geometry.

Constant Camera Motion

One of the main objectives of SHOCam is to maintain a constant camera motion while orbiting an object. Critical situations can occur around concave corners or holes, where the closest point suddenly jumps from a polygon to a far one. Existing methods propose to interpolate the camera direction and positions, but they do not work sufficiently well, as explained above.

Here we propose to compute new camera positions in iterative steps. In each step, we start with the traditional HoverCam algorithm. As this algorithm does not guarantee that the distance between the new viewpoint position and the last one (d) is equal to the length of the desired displacement distance ($|V|$), we repeatedly iterate HoverCam computations until $d >= |V| - \alpha * |V|$, where α corresponds to the maximum percentage of variation in displacement distance we accept.

For this, each new computation uses a new displacement vector, whose size and orientation depends on the previous computation result: its orientation is orthogonal to the computed viewpoint direction and the size is equal to $|V| - d$. Here is the algorithm:

```
Initial Parameters
      - D: orbiting distance
      - CC: Current Camera position
      - V: current displacement vector
      - $\alpha$: maximum percentage of accepted variation

Algorithm
1 – Compute the New Camera position (NC) with
HoverCam(CC,D,V)
2 – Compute d =distance(CC,NC)
3 – If $d >= |V| - \alpha*|V|$ or iteration ==4:
          then stop algorithm
    else:
         Compute new V from NC
         $|V| = |V| - d$
         CC = NC
         Go back to step 1
```

To avoid the camera penetrating into the orbited object, $|V|$ (the initial displacement vector length) should never exceed the orbiting distance. Our algorithm guarantees that the distance that the camera moves between two consecutive cameras is always in $[|V| - \alpha*|V|,|V|]$ (see Figure 3). As with all iterative algorithms, the algorithm might not terminate. If the camera does not move, we stop after 4 steps, which corresponds to the situation that the camera is in the center of a spherical cavity and the displacement distance is close to its radius. This can only happen if the user increased the target distance after navigating into a cavity through a smaller gap (or started there). In this case we issue an auditory alert and reduce the displacement distance temporarily by 10% to keep the user mobile.

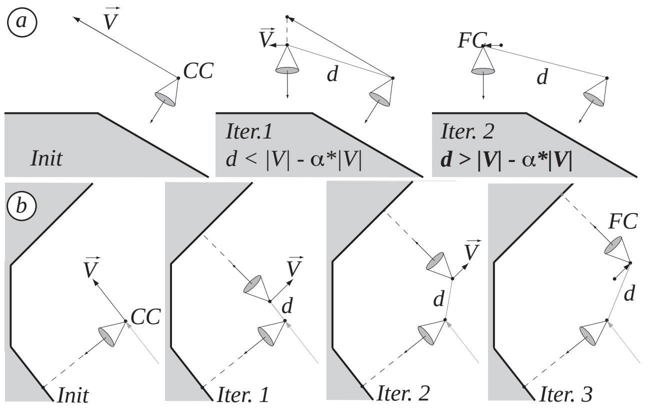

Figure 3. Iteration for SHOCam. a) orbiting around a convex surface. For clarity of illustration, the length of *V* is here higher than the distance between CC and the surface. b) orbiting on a concave surface.

The computation of the closest point is the most expensive part of the algorithm. One solution is to compute this analytically. As the closest point has to be on a polygon of the orbited object, we compute the distance between the camera and each polygon of the object and take the minimum of all distances. For more complex scenes, another solution is the use of an image-based approach, *e.g.*, through the cubemap method [13], which identifies the closest point from the depth buffer. The 3D position of the pixel with the smallest depth value together with the camera position defines a 3D ray. Intersecting that ray with the scene yields the closest point (up to discretization errors).

This approach provides a regular discretization of the theoretical path of the camera. This then ensures a constant mapping between user interaction and the theoretical camera displacement around the object. Yet, as with all discretization algorithms, this is not enough for guaranteeing G1 continuity of the camera path. G1 continuity is an appropriate criterion for smooth visual feedback and optical flow. In the next section we present how we generate a G1 camera path.

Smooth Interpolation of Camera Paths

The different camera positions computed by the iterative algorithm are spread regularly along the theoretical camera path. However, using these positions directly does not guarantee G1 continuity for the camera path. In Figure 3b we see that if the camera jumps from its current position

(*CC)* to the final one (*FC*), the displacement is not G1 continuous relative to the previous one. Consequently, we compute a smoother camera path by interpolating positions from a *set* of pre-computed local cameras around the main camera (see Figure 4). To simplify the description, we first describe navigation of a camera in 2D along a 1D path.

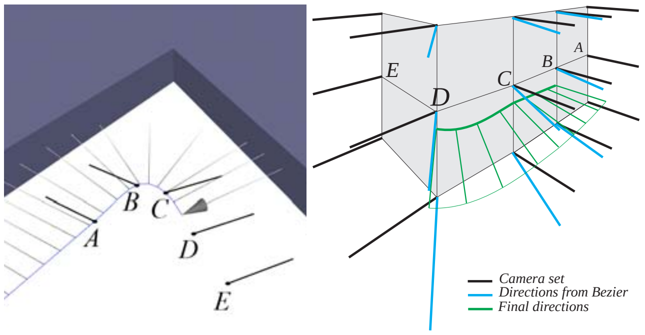

Figure 4. Left: in this 2D example, the final path of the camera is a Catmull-Rom spline, computed from the local camera set defined by five pre-computed camera positions (*A* to *E*). When the current camera reaches *D*, a new camera position (*F*) is computed and the set adjusted to contain *B* to *F*. Right: example camera direction computation via two dimensional interpolation. As camera direction does not change vertically within the camera set, we use 1D cubic Bezier interpolation at each of the three horizontal sub-sets (blue vectors).

When orbiting is initiated, five new camera positions are computed, two in the forward direction and two backwards. The length of the displacement vector for computing these new positions has to be higher than the final camera velocity. We describe below how we choose these lengths. With these five camera positions, we compute the final camera path with spline interpolation. The interpolation should satisfy the two following constraints: (1) as our iterative viewpoint computation method provides “correct” camera positions, where the distance between the camera and the object surface is constant, the final camera path should pass through these positions, and (2) the interpolation should guarantee G1 continuity of the path. For simplicity, we choose the *Catmull-Rom* interpolation spline. Figure 4-left illustrates the resulting path.

Once the first camera set is computed, the camera can then easily travel in one dimension along the spline, with a direct mapping between the user’s device displacement and main camera position. A new camera position for the set is computed only when the main camera reaches the next forward or backward position, which then triggers the update of the camera set. For efficiency, the computation of the new camera set can be done in a separate thread, while the camera is still traveling on the current set (see the discussion section).

The length of the displacement vector *V* used for computing the camera set has to be higher than the current camera velocity (*v*). We use Mackinlay’s approach [12], *i.e.*, the camera velocity depends on the orbiting distance (*D*). Also, to guarantee that the camera never penetrates the surface of the orbited object, we keep the camera speed in the interval [|*v*|, *D*]. SHOCam offers then a large amount of possible camera paths, illustrated in Figure 13.

We generalize our new method to (horizontal) 3D orbiting by computing two more horizontal camera sets (see Figure 4-right), one above and one below the current set (relative to its current coordinate frame). The vertical distance between the sets is equal to the distance for the horizontal set (using the same camera velocity limits as above [12]). The final camera then navigates on this 2D camera surface. Its position is defined via 3D Catmull-Rom interpolation from the whole camera set. When the camera moves far enough, additional cameras in the corresponding direction are computed in separate threads. As the next set overlaps the current one, only a few cameras need to be computed at a given time. Vertical orbiting is handled analogously.

Smooth Interpolation of Camera View Directions

Interpolating the previously presented camera set can provide smooth changes in view directions. However, in order to maintain a comfortable optical flow, the change in look direction has to be minimized at each time step, and spread along the movement when the camera reaches a corner. Thus, we cannot use interpolating splines for direction, as they compute a spline that passes through its control points. Consequently, we use cubic Bezier interpolation, which takes the cameras in the camera set as control “directions”. Cubic Bezier interpolation requires four control points and this determines how many cameras are necessary around the current camera. We then compute three directions per camera set. These directions correspond to the B, C and D positions in Figure 4-left. In general, final view directions will rarely be exactly equal to the initial view directions in the camera set. In other terms the camera will not always look precisely at the closest point. Yet, this flexibility permits us to minimize of the variations in both camera positions and view directions, which in turn reduces the variations in the optical flow. Figure 4-left illustrates this, and one can notice the “anticipation” of the corner.

As SHOCam does not vary the camera direction “vertically” (along the camera up vector), the interpolation for camera directions only has to respect cubic Bezier characteristics in its “horizontal” dimension. The 3D camera set is then made of three horizontal 2D “sub”camera sets, and a cubic Bezier is applied on each of them independently (see Figure 4-right) before computing the final direction. The latter is then a simple Catmull-Rom interpolation from the sub-sets’ directions.

Constraining the Camera Path

Assuming a horizontal camera movement, one issue of the existing methods presented above is the vertical shakiness implied by the erratic positions of the closest point on detailed surfaces (see Figure 2c). This may also imply vertical changes in the camera path, even if the user makes a pure horizontal movement. To avoid this, we constrain the positions of each camera of the set to the plane defined by its local position and its up vector. This method constrains the final camera movement and also further reduces changes in the optical flow. As illustrated in both the numerical and the user evaluation below, the added stability in terms of camera direction makes a significant difference.

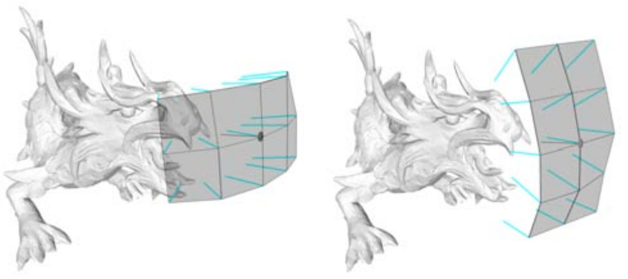

Figure 5. Navigation with horizontal or vertical camera sets. The camera navigates on a 2D shape, and its position is a Catmull-Rom interpolation from one of the camera sets, depending on the initial movement of the user.

Orbiting in both directions

With the camera set shown in Figure 4, both horizontal and vertical movements are constrained by the orbiting distance. Yet, the view direction is always horizontal. Thus the view direction will never be rotated down to see, *e.g.*, the top of the dragon's head (Figure 5). We thus propose a new interaction technique that uses two camera sets, for horizontal and vertical movements respectively (Figure 5 left and right). For this we initially compute both camera sets (note that the 3x3 interior camera *positions* are shared). When the user starts to interact, we analyze the initial path of the user to determine if the user is moving horizontally or vertically and then use only the corresponding set. Then, if the user has been navigating in the orthogonal direction previously, the camera direction may not face the surface. Thus we blend the camera direction in the first second of the interaction between the direction computed from the camera set and the old camera direction.

Orbiting in Multi-Objects Scenes

The presented method is efficient and generates consistent camera paths for single object orbiting. However, orbiting an object in close proximity to other objects can potentially cause problems as the user may lose sight of the orbited object if these objects can influence the camera path. Here, a camera path should respect two main constraints:

- The camera should only focus on the orbited object. For example, when orbiting one particular tree in a forest, the camera should never face another tree.
- When displaying other objects transparently is not an option, the camera should never permit the user to move into objects. Moving the camera into another object also leads to loss of the context of the orbited object.

Based on these constraints, we see that the camera orbiting behavior (both camera positions and view directions) might need to be influenced by other objects in some limited manner in some situations.

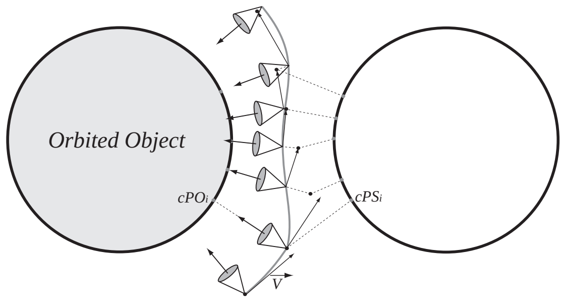

Figure 6. The orbited object is always the closest. The camera moves along its view direction and never penetrates into the rest of the scene.

Due to the dependency on the availability of transparency, there are two fundamental choices, based on the task constraints and/or rendering capabilities.

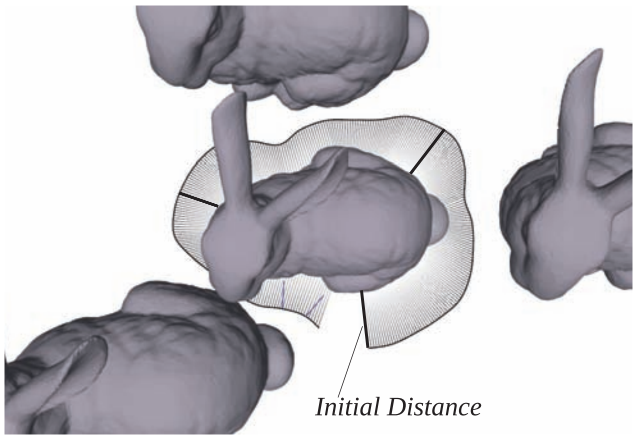

Figure 7. Illustration of the adaptation of the orbiting path to other geometry in SHOCam. The camera always faces the object along the path. Due of the other bunnies, the camera distance varies between the initial distance to the orbited bunny (thicker black bars) and smaller distances.

1) In an observation/inspection task, the user mainly concentrates on the orbited object. Thus the camera should stay at a constant distance to the orbited object. Here, the easiest solution is to keep the initial orbiting path and to make other objects transparent when necessary, *i.e.*, when the camera moves into them or when parts of them appear in the field of view of the camera. For this option, we use Ortega et *al.*'s approach [16]. At each camera motion step, we compute both the closest point on the orbited object (*cPO*) and the closest point on the rest of the scene (*cPS*). If *cPS* is the closer one and is in the camera field of view, we compute two camera frustums: one from the camera's near plane to the *cPS*, and a second from the *cPS* to the camera's far plane. The first frustum is used to render the scene in a texture, which is transparently superimposed onto the final rendering of the second frustum. This method guarantees that only parts of the scene that could affect the view of the object are shown transparent.

2) In navigation tasks, such as orbiting around a wall corner in a First-Person-Shooter game, constraints imposed by the environment may play a significant role for navigation. There, the camera path should take the whole scene into account. The transparency method is not sufficient in such situations and should not be used as it radically changes the visual appearance of the scene and also leads to a potential loss of visual context. Moreover, in many situations it is undesirable if the camera can move into other objects. A classic example is a game where a user inside a wall is able to shoot others without ever getting hit.

```
-Compute the closest point on the orbited object (cPO) and
    on the whole scene (cPS).
-if dist(camera,cPO) < dist(camera,cPS):
    -if the previous cPS was closer:
        -move camera backward along its line of sight,
            keeping dist (camera,cPS) >= dist (camera,cPO)
    -else:
        use normal orbiting behavior
-else:
    -move camera frontward along its line of sight,
            until dist (camera,cPS) > dist (camera,cPO)
```

Figure 8. Pseudo-code for adapting the orbit to other objects in the scene.

Here we propose a new solution that uses both the closest distance to the orbited object *cPO* and also to the rest of the scene *cPS*. According to these distances, the camera then moves along its view direction to ensure that the camera always faces the orbited object, and never penetrates into the rest of the scene. The amount of translation of the camera is dynamically adjusted so that the orbited object is always the one that is the closest to the camera. According to our algorithm, the camera moves forward when the rest of the scene comes closer, and backwards to the initial distance when the distance to the rest of the scene increases again. **Figure 8** shows pseudo-code for this algorithm.

Figure 6 illustrates the resulting camera path. As camera speed is directly proportional to camera distance in the rest of our system (see above) we also reduce the speed of the camera whenever it gets closer to the orbited object. Figure 7 represents such a path in between 3 bunnies.

GROUP ORBITING

Another interesting issue in orbiting behaviors is that small objects close to large objects or other far away objects can negatively influence orbiting behaviors. For example, when orbiting a planet that has close by satellites, the method presented above can force the camera to be "squeezed" between the satellite and the planet. But orbiting the large object can also have the camera orbit *into* the satellites.

To address this issue, we use the bounding volume hierarchy for the whole scene to automatically determine the group to orbit with a criterion that adapts itself to the scale of the viewer. When the user initiates an orbiting action, the system first determines the current projected size of the bounding volume of the chosen object/group. We then use two user-configurable thresholds, with empirically determined defaults of 30% and 15% of the screen area, to adjust the group. Three scenarios are possible:

a) The selected object/group projects to a size that is too large (greater than the larger threshold). In this case, the system searches for smaller groups whose projection size is smaller than this threshold (if they exist) to be included in the orbited group (see Figure 9).

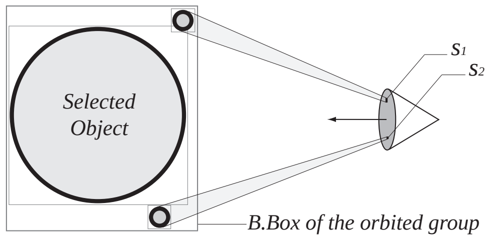

Figure 9. Orbiting objects with a wide disparity in size. As both satellites project to a small size on the screen (*s1,s2*), orbiting around either of them from this distance will result in the camera orbiting around the whole group instead.

b) The selected object/group projects to a small size on the screen (smaller than the smaller threshold). In this case, we assume that most of the time the user would like to orbit a larger group, if available (see Figure 10). The system then searches for progressively larger groups until it finds one that is larger than the smaller threshold. Sometimes the user would like to orbit a single object. This still requires that the object is large enough to be (easily) selectable by the user, which may require moving closer.

c) If the user does not explicitly select an object and starts orbiting, the system detects the closest object to both the camera position and the view vector, and then applies the appropriate above logic. If the viewer is too close to geometry to identify an object that fits into the view frustum, the first object along the view vector is chosen.

Once a group is selected, explicitly or automatically, the system then orbits around that whole group, *i.e.*, around all the objects in the group simultaneously. Figure 9 illustrates how the SHOCam method adapts the orbiting path in the presence of other objects around the viewer. Finally, Figure 10 illustrates how the orbit is affected when the method orbits around a group of objects. While the above-mentioned heuristic thresholds work well for all examples shown here, they may have to be adapted to other use cases.

NUMERICAL EVALUATION

For this evaluation, and for the accompanying video, we implemented HoverCam and SHOCam in Python, with PyOpenGL and PyCollada. The computation of the closest point is done analytically, using several functions in C. As the *camera set* used in SHOCam needs intermittently the computation of many camera positions at the same time, we parallelized the computation of "next" sets. These computations are done in the background, while the user is manipulating the viewpoint within the current set. For our implementation of SHOCam, we set $\alpha = 0.9$ and length of V (the vector used to compute the *camera set*) equal to half the distance from the camera to the object surface.

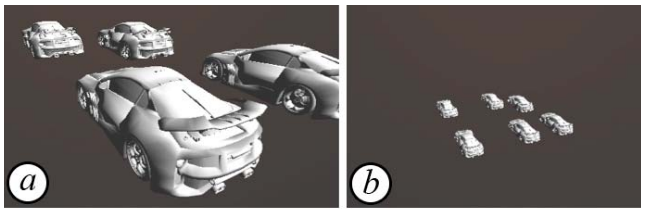

Figure 10. Group orbiting illustration: *a*) the user is fairly close to a car and likely wants to orbit around this particular one; *b*) the user is further from any individual car. Here our algorithm automatically treats the group of six cars as a "single" object and orbits around the entire group.

Optical flow

The difference in the quality of the camera paths generated by SHOCam is easily visible in terms of the camera paths and view directions in the figures of this paper as well as in the accompanying video. As additional support, we perform here a numerical comparison of HoverCam and SHOCam.

Figure 11 first illustrates the variations in viewpoint direction for a path around 3 bunnies. The curve represents the angles (in degrees) between each consecutive camera direction vector. Two observations can be made: (1) the path is 38.4% longer with HoverCam, and (2) HoverCam suffers from a lot of abrupt variations in viewpoint direction while SHOCam is much more stable. As HoverCam's viewpoint is interpolated in concavities, this is another source of velocity variations and causes the longer path. The abrupt variations are due to the very detailed geometry.

Figure 11 also illustrates the variations of the viewpoint velocity along the same path. The curve represents the distances between the viewpoint positions. Compared to the previously mentioned curve, one more observation can be made: Hovercam is prone to abrupt "braking". Indeed, 30% of viewpoint movements are only rotations. Also, the standard deviation (stdev) of tangential speed is 42.07% with HoverCam vs. 0.39% with SHOCam.

To summarize, the path is smoother with SHOCam and the rotational speed has also smaller variations. The shown results are representative for all the models presented here.

Computational Complexity

The bottleneck of all considered methods is the computation of closest points (*CCP*), regardless if computed analytically or image-based. Thus, comparing the computational complexity of each method by investigating the number and the frequency of *CCPs* is more objective than comparing times. For a final camera position, HoverCam uses a single *CCP* most of the time, and sometimes no *CCP* (when interpolation is used). SHOCam uses no *CCP* most of the time, as the camera is interpolated, and only uses *CCPs* to update the *camera set*. However, SHOCam always computes less *CCPs* than HoverCam. For example the dragon orbit (around 414 viewpoints for SHOCam and 635 for HoverCam), involve 225 *CCPs* for SHOCam vs. 450 for HoverCam (77% more).

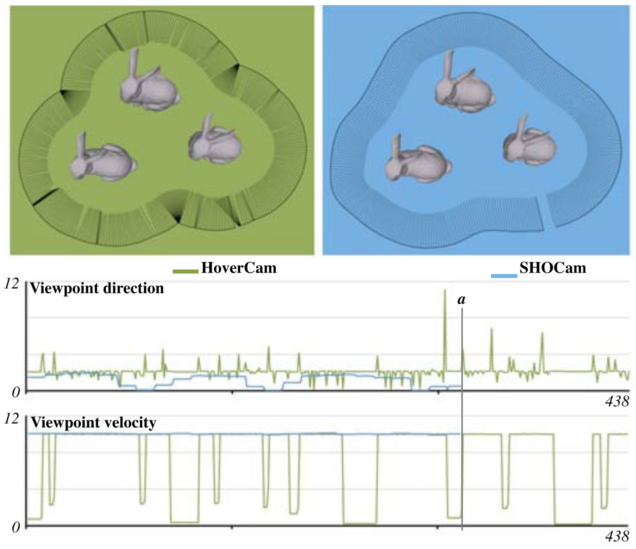

Figure 11. Variations of viewpoint direction (in degrees) and viewpoint velocity vector (in arbitrary 3D distance) with HoverCam and SHOCam for a path around the bunnies. Point *a* marks the end of SHOCam path, HoverCam needs more steps and velocity variations are significantly higher .

SHOCam reuses cameras when updating the *camera set*, which decreases "bursts" in computation, which is further "hidden" via multi-threading. As most cameras are interpolated, this yields similar frame rates for both techniques and most all the models presented in Figure 12.

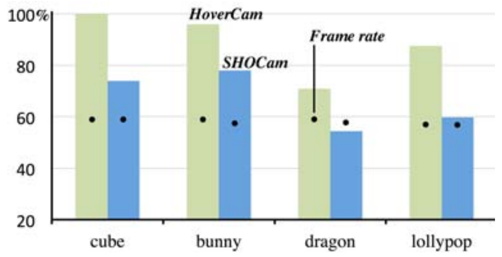

Figure 12. Percentage of *CCP* relative to total camera positions during one orbit around 4 models (three models shown in other figures, the single cube not shown). HoverCam always needs more *CCPs*: from 17% to 28% more. Dots show the frame rate (in *Hz*) for each technique.

In all the presented paths, with SHOCam, the number of *CCPs* for a single camera of the *camera set* is never higher than three. Our future investigations will try to link this number, the *CCPs* and the frame rate with the scene complexity in order to evaluate how HoverCam and SHOCam deal with very complex scenes.

USER EVALUATION

In order to evaluate perception of any potential differences between the Hovercam and SHOCam, we conducted a user evaluation. The goal was to detect if (1) the difference between the two techniques is perceivable by users, and if (2) this makes a difference in orbiting manipulation. The evaluation was conducted on a Wacom CINTIQ, *i.e.*, a touch screen with 1920x1200 pixels.

Protocol

We presented three objects to 14 participants (between 22 and 38 years old, 6 women). Each participant orbited around each object with HoverCam and with SHOCam. We used counter-balanced presentation to cancel potential learning effects. We first introduced each technique with an automatically computed orbit path and then let users orbit interactively. For the latter, we mapped 2D touch movements to up/down and left/right movements of the viewpoint. We used (1) a Stanford bunny (Figure 7) as a standard and “compact” object, (2) a Stanford dragon (Figure 1) as a more complex object (different aspect ratio and with a lot of concavities and convexities) and (3) a lollipop star (Figure 13) as an object that should stress both techniques. In the interactive part, people were free to explore the objects. For the star-lollipop we asked users to turn around the object as fast as possible, and to judge which technique seemed more efficient. After the automated and interactive exploration phases, participants filled out a questionnaire. They first rated their perception of each technique, followed by more specific questions.

Results

All participants perceived and were disturbed by the “sudden jumps” and “shakiness” of the camera direction with HoverCam. No one complained about this for SHOCam. 11 participants estimated that they were faster in exploring the objects with SHOCam. They especially mentioned the lollipop exploration, where the viewpoint would get temporarily “stuck” in concavities, but also for the dragon. The bunny did not suffer from this issue. Three participants did not perceive a difference in interaction velocity between the two techniques. 9 participants perceived cybersickness with HoverCam, and one complained about both techniques. Cybersickness was mainly perceived when exploring the dragon. All but one of the participants were satisfied with the SHOCam interaction control, but only 4 with HoverCam. Indeed, 11 participants complained about the “unpredictable” behavior of HoverCam, especially with the dragon because of (1) visual jumps and (2) because HoverCam restricts user input to a single dimension during interpolations. Indeed, when the user is horizontally orbiting around the dragon, the closest point can jump between the head and a foot. This makes the optical flow suddenly vertical, creating a mismatch between user movement and viewpoint behavior.

All but two participants preferred SHOCam for all kinds of orbiting tasks, such as navigation around, global overview, and quick exploration. Two participants noted that each technique has its own advantages. Indeed, they felt that HoverCam would be better for “precise” exploration. As we were surprised by this comment, we asked for more details and identified that this is side effect caused by the “sticky” effect of HoverCam around corners or salient areas. Indeed, as the viewpoint lingers on such points, users focus more attention on them. However, this only happened on specific points (convexities), and was only determined by the geometry and not the user. The interaction asymmetry and the fact that it is somewhat unpredictable make us believe that this is not an inherent benefit of HoverCam.

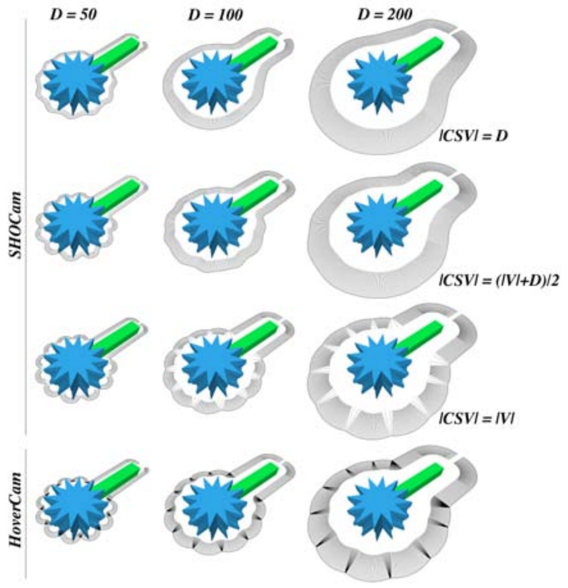

Figure 13. Camera paths that illustrate HoverCam and SHOCam behaviors around a “star-lollipop” (used in the user evaluation). *D* (in 3D arbitrary length) is the distance between the camera and the object’ surface (lollypop length is about 1600). Length of *camera set* vector (*CSV*) varies from |*V*| to *D*. When |*CSV*| = |*V*|, SHOCam’s behavior is close to HoverCam, but without interpolation in concavities and without vertical variations in camera directions.

DISCUSSION

To illustrate the camera paths that SHOCam generates, we show a side-by-side comparison of orbits in Figure 1. For better clarity, the viewpoint has been controlled only horizontally: the viewpoint path is then horizontal. One can see the notable differences in the way the camera behaves around convex vertices and/or concavities. In specific cases, *e.g.*, when the camera is very close or very far from the object surface, the HoverCam and SHOCam trajectories are very similar. Yet, SHOCam will still require less CCPs than HoverCam, *i.e.*, be more efficient.

In the top part of Figure 13 the camera does not directly face the flat parts of the green rod, close to the star part. Participants did not notice this during the evaluation. However, if the user needs to precisely observe these locations, one should consider alternatives. We will investigate this in future work.

In the presence of other geometry, the orbiting behavior of SHOCam around objects is also smoother. We initially debated to have SHOCam permit the user to get much closer to geometry behind the camera. Yet, this is a double-edged sword. The closer one gets to geometry behind the camera, the more it will influence the path of the camera. Imagine moving with your back to a wall that has spikes. The path would then be very jagged. Thus, we believe that SHOCam's approach is an interesting "middle ground" (both in approach and camera path) relative to existing approaches. Still, for inspection/drawing tasks where the distance to the object has to stay constant we recommend an approach where the camera can orbit into other objects. In this case we recommend that parts of objects in front of the camera be rendered transparently [16].

One of the nice features of the group orbiting method is that it obviates the need to select an orbiting center in most situations. If the user moves further away, the system automatically adapts to the larger context and orbits around larger groups. Conversely, when moving closer, the system orbits around smaller groups or individual objects. Having said that, we recognize that our heuristic may fail and may not always correspond to the intentions of the user. Hence, we believe that this approach should be used judiciously and preferably when the user has not set an explicit entity to orbit and/or when the user has moved enough so that a previously set orbiting center is not relevant, *e.g.*, because it is now invisible or too small to be practically useful.

CONCLUSION

We presented SHOCam, a new orbiting algorithm that guarantees simple, safe and visually attractive camera paths for orbiting around 3D objects in 3D scenes. Compared with existing methods, SHOCam provides a more consistent mapping between the user's interaction and the path of the camera by substantially reducing variability in both camera motion and look direction. Moreover, we presented a new method that prevents the camera from orbiting into other objects, making the visual feedback, and so the user experience, more pleasing and also less error prone. Finally, we present a new method for automatically determining the group of objects to orbit.

AKNOWLEDGMENTS

This work was partially funded by the FUI project 3DCI (AAP14), ANR ISAR, the GRAND NCE, NSERC, and Persyval Labex.

REFERENCES

1. Argelaguet F., Adaptive navigation for virtual environments, *Proc. IEEE 3DUI 2014, pp. 123-126*
2. Bae S., Balakrishnan R., Singh K., ILoveSketch: as-natural-as-possible sketching system for creating 3d curve models. *Proc. ACM UIST 2008, pp. 151-160*
3. Burtnyk N., Khan A., Fitzmaurice G., Kurtenbach.G., ShowMotion: Camera motion based 3D design review. *Proc. ACM I3D 2006, pp* 164-174
4. Christie M., Olivier P., Camera control in computer graphics: models, techniques and applications. *Proc. ACM SIGGRAPH ASIA Courses 2009, Art. n.3*
5. Decle F., Hachet M., Guitton P., Scruticam: Camera manipulation technique for 3D objects inspection. *Proc. IEEE 3DUI 2009, pp. 19-22*
6. Dorado J.L., Figueroa P. A., Ramps are better than stairs to reduce cybersickness in applications based on a HMD and a gamepad, *IEEE 3DUI 2014, pp. 47-50*
7. Fitzmaurice G., Matejka J., Mordatch I., Khan A., Kurtenbach G., Safe 3D navigation. *Proc. ACM Symposium on Interactive 3D Graphics 2008*, pp. 7-15
8. Hachet M., Decle F., Knoedel S., Guitton, P., Navidget for easy 3d camera positioning from 2d inputs. Proc. *IEEE 3DUI 2008*, pp. 83-89.
9. Hanson, A. J., Wernert, E. A. Constrained 3d navigation with 2d controllers. Proc. *IEEE VIS 1997*, pp 175–183
10. Khan A., Komalo B., Stam J., Fitzmaurice G., Kurtenbach G., HoverCam: Interactive 3D navigation for proximal object inspection. *ACM I3D 2005, pp. 73-80*
11. Kopper, R., Ni T., Bowman D.A., Pinho M., Design and evaluation of navigation techniques for multiscale virtual environments, *Proc. IEEE VR 2006*, pp. 175-182.
12. Mackinlay J. D., Card S. K., Robertson G.G., Rapid controlled movement through a virtual 3D workspace, *Proc. ACM SIGGRAPH 1990, pp. 171-176.*
13. McCrae J., Mordatch I., Glueck M., Khan A., Multiscale 3D navigation. *ACM I3D 2009, pp.7-14*
14. Moerman C., Marchal D., Grisoni L., Drag'n Go: Simple and fast navigation in virtual environments, Proc. *IEEE 3DUI 2012*, pp. 15-18
15. Nieuwenhuisen D., Kamphuis A., Overmars M.H., High quality navigation in computer games. *Science of Computer Programming* 67, no. 1 (2007): pp. 91-104.
16. Ortega M., Vincent T., Direct drawing on 3D shapes with automated camera control. *Proc. ACM CHI 2014, pp. 2047-2050*
17. Oskam T., Summer R.W., Thuerey N., Gross M., Visibility transition planning for dynamic camera control. Proc. *ACM Eurographics 2009*, pp.47-57
18. Phillips C.B., Automatic viewing control for 3D direct manipulation. *Proc. ACM I3D 1992, pp.71-74*
19. Schmidt R., Khan A., Kurtenbach G., Singh K., On expert performance in 3D curve-drawing tasks. *Proc. ACM SBIM 2009, pp. 133-140*
20. Tan D. S., Robertson G.G., Czerwinski M., Exploring 3D navigation: Combining speed-coupled flying with orbiting. *Proc. ACM CHI 2001*, pp. 418-425.
21. Trindade D.R., Raposo A.B., Improving 3D navigation in multiscale environments using cubemap techniques. *Proc. ACM SAC 2011, pp. 1215-1221*
22. Ware, C., Fleet, D., Context sensitive flying interface. *Proc. ACM I3D 1997*, pp. 127-130
23. Zeleznik R., Forsberg A., UniCam – 2D gestural camera controls for 3D environments. *Proc. ACM I3D 1999, pp. 169-173*
24. Zhang X., A multiscale progressive model on virtual navigation. *International Journal of Human-Computer Studies* 66(4), 2008, *pp. 243-256*