# HoverCam: Interactive 3D Navigation for Proximal Object Inspection

Azam Khan, Ben Komalo, Jos Stam, George Fitzmaurice, Gordon Kurtenbach

Alias
210 King Street East, Toronto, Ontario, Canada
{akhan | bkomalo | jstam | gf | gkurtenbach }@alias.com
www.alias.com/research

## Abstract

We describe a new interaction technique, called *HoverCam*, for navigating around 3D objects at close proximity. When a user is closely inspecting an object, the camera motions needed to move across its surface can become complex. For tasks such as 3D painting or modeling small detail features, users will often try to keep the camera a small distance above the surface. To achieve this automatically, HoverCam intelligently integrates tumbling, panning, and zooming camera controls into a single operation. This allows the user to focus on the task at hand instead of continuously managing the camera position and orientation. In this paper we show unique affordances of the technique and define the behavior and implementation of HoverCam. We also show how the technique can be used for navigating about data sets without well-defined surfaces such as point clouds and curves in space.

**Categories and Subject Descriptors:** I.3.6 [Computer Graphics]: Methodology and Techniques – Interaction Techniques; H.5.2 [Information Interfaces And Presentation (HCI)]: User Interfaces – Interaction styles, Input devices and strategies.

**Additional Keywords and Phrases:** interaction techniques, camera controls, 3D navigation, 3D viewers, 3D visualization.

## 1 Introduction

The most commonly used operation in 3D computer graphics and animation software is camera movement. Users often move the camera to help them sense the 3D properties of a model or animation or while performing modifications. When working at close proximity to an object like, for example, when painting details on an object, the camera must often be moved to see neighboring surfaces. However, despite the heavy usage of camera tools in 3D content creation software, the industry standard zoom, pan, and tumble tools have been the primary camera controls offered to users for over a decade.

For keyboard and mouse based user interfaces, interactive 3D camera control is fundamentally difficult because the task involves controlling the six distinct degrees of freedom (DOF) (translation x,y,z and rotation $\alpha,\beta,\chi$) of the virtual camera with just two DOF of mouse input. The simplest approach is to assign the two DOFs of the mouse to two different DOFs of the camera, at different times. However, more sophisticated approaches that emulate real world behaviors, or better match the task at hand and/or the skills of the user, have largely replaced the simple approach. For example, the industry standard zoom, pan, and tumble tools reflect typical methods of controlling physical camera from the film production industry. Even more sophisticated camera control techniques take into account information about the scene. For example, 3D video games often have a *walking* metaphor of camera motion. This metaphor suggests many things: there is a ground plane, the viewpoint is somewhat above the ground, camera rotation is egocentric, there is notion of which way is “up”, etc. These constraints simplify camera motion from a general 6 DOF problem to almost a 2 DOF problem. Further constraints, such as collision detection, prevent the camera from passing through walls, characters, and objects in the scene. This entire set of constraints is needed to convey the *walking* camera metaphor.

The camera metaphor we explore in this paper is navigation around 3D objects at close proximity. For this task, we would like to move around the object while maintaining a fixed distance from the surface and while keeping the object roughly centered in the field of view. We call this metaphor *object inspection*. As with the *walking* metaphor described above, we attempt to use as many constraints as possible, given by the context of our metaphor, to simplify the number of controls needed to navigate through space during typical object inspection.

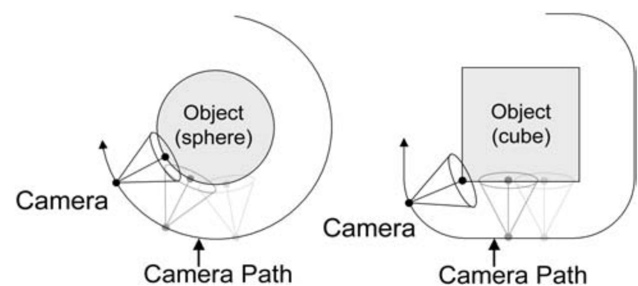


Figure 1. Desired HoverCam motion over a sphere and a cube (shown in profile).

Copyright © 2005 by the Association for Computing Machinery, Inc.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions Dept, ACM Inc., fax +1 (212) 869-0481 or e-mail permissions@acm.org.
© 2005 ACM 1-59593-013-2/05/0004 $5.00

## 2 HoverCam

Figure 1 shows the type of camera motion behavior we would like for a simple sphere and cube. Note that given the constraints of (1) keeping the camera a fixed distance from the surface and (2) relatively normal to the surface effectively creates a shell around the object being observed, on which the camera can be positioned. Around curved surfaces, the camera follows the surface but at a fixed distance. Around corners, the camera turns to smoothly move toward a neighboring surface. Finally, around flat surfaces, like the side of the cube, the camera pans as expected to keep the underlying surface facing the viewpoint.

Camera paths like these can be achieved using the traditional separate pan, zoom, and tumble tools but at the cost of constantly switching between the three tools. Also, as these tools do not typically perform any collision detection, users may end up in awkward locations inside the object, looking away from the object, or at great distances for the object. For example, in Alias' Maya software, the system is placed in camera mode by holding down the alt-key. Dragging with the left mouse button then tumbles the camera (rotates about the current look-at point). Dragging the middle mouse button pans the camera (translates the eye and the look-at point) and dragging with both left and middle mouse buttons performs zooming (moving the eye toward or away from the look-at point). Releasing the alt-key stops the camera tool and reselects the user's previous tool.

A smooth camera path around the outside of an object is simply not achievable with these separate tools. To keep a point of interest on the surface of the object near the center of the view, the user must always overshoot, switch tools, correct the view with another tool, overshoot again, and so on.

### 2.1 Basic HoverCam Algorithm

A smooth camera path can be achieved with a trajectory algorithm loosely based on the model of a satellite orbiting an object with a gravity field (see Figure 2). The steps performed are:

(a) apply user input to the *eye* point $E_0$ (current camera position) and *look-at* point $L_0$, to create $E_1$ and $L_1$,

(b) search for the *closest* point C on the object from the new eye position $E_1$,

(c) turn the camera to look at C, and,

(d) correct the distance $\delta_1$ to the object to match the original distance to the object $\delta$ to generate the final eye position $E_2$.

(e) clip the distance traveled (discussed in Section 2.5).

This algorithm, in effect, selectively combines the operations for zooming, panning, and tumbling during a single mouse drag. This has the advantage that HoverCam only requires a single button mouse, pen-press, or a single finger press on a touch screen to apply camera motion. In contrast, as mentioned earlier, standard zoom, pan, and tumble tools typically require multiple buttons to switch between the operations for zooming, panning and tumbling. Furthermore, to achieve HoverCam motion with the traditional separate tools would require ongoing switching of the tools to continually correct the camera motion to follow the surface.

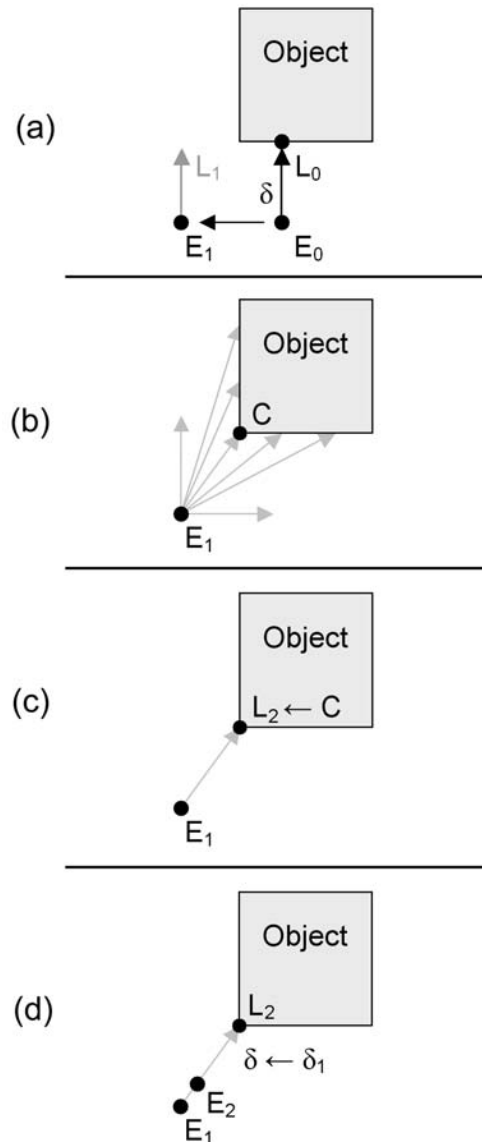


Figure 2. Basic Camera Update Rules. (a) move eye based on user input, (b) look for C, closest point on object, (c) turn camera to C, and (d) correct distance.

Using HoverCam has the feeling of hovering above the object. Figure 3 shows two sets of screen images showing the user's perspective as HoverCam is being used to inspect a cube and a cylinder, maintaining a consistent scale and distance from the object. Note how HoverCam pans on the side of the cube, and turns about the corner of the cube. Also note that on the cylinder, the camera pans along the shaft, turns smoothly to the end disc, and pans across the disc.

To highlight the difference between traditional center-based camera motion and surface-based camera motion, see the example of motion about a cylinder in Figure 4. With the simple traditional tumble, the rotation about the cylinder would have placed the camera inside the object. However, with HoverCam, moving to the right pans the camera until it can rotate about to continue panning across the end disc.

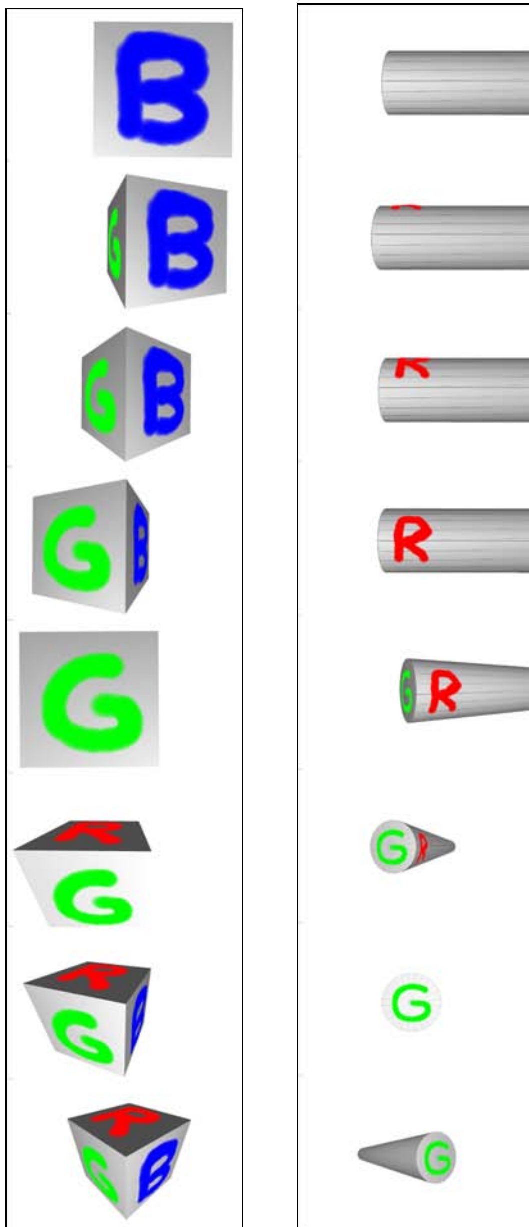

Figure 3. HoverCam around a cube and a cylinder, from the point of view of the user.

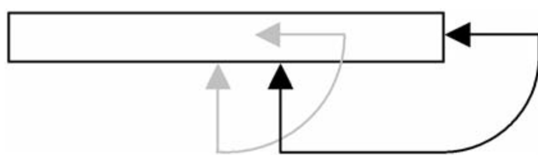

Figure 4. Simple Rotation versus HoverCam: The grey path shows how a simple rotation about the center of a cylinder leaves the camera within the object. However, HoverCam moves the camera along the cylinder and only rotates when turning about the end of the shaft (black path).

## 2.2 Blending Camera Techniques

As HoverCam would normally be used together with traditional freeform navigation tools, we have designed HoverCam to interoperate between the various camera techniques in a fairly seamless way. Freeform camera motion allows the user to navigate to any point in space and to face in any direction. For specific surface-based tasks like 3D painting or sculpting, HoverCam provides a subset of this freedom with the benefit of following the surface. Switching from HoverCam to a freeform camera could simply be invoked by clicking on a tool icon or by a key press. However, switching from a freeform camera to HoverCam may cause an abrupt reorientation and reposition of the camera because an initial search may find a result quite far from the current view. Two methods are used to ease this disruption. In the case where a freeform camera approaches an object from a significant distance, a field of influence around the object specifies how strongly the HoverCam motion is linearly interpolated with the freeform motion. In the case where a freeform camera is already very close to an object (fully within the HoverCam field), motion clipping (as discussed in Section 2.5) is applied to smoothly transition to HoverCam motion.

Layers of HoverCam influence around each object are automatically generated (see Figure 5). The outer layer is quite far from the surface and specifies a *field of influence*. Once a freeform camera enters this field, the HoverCam camera is weighted together with the freeform camera so that it will be sucked towards the outer limit of the orbit distance (see Figure 6). Once the camera is fully controlled by the HoverCam algorithm, it remains so until the user switches to another navigation method. In practice, we have found it helpful for the user to be able to specify the distance between the surface and the camera. In our current implementation, the mouse-wheel is used to zoom in or out to specify a new fixed distance to the HoverCam algorithm. The HoverCam camera orbit distance will always be between the inner limit and the outer limit unless the user zooms out beyond the field of influence.

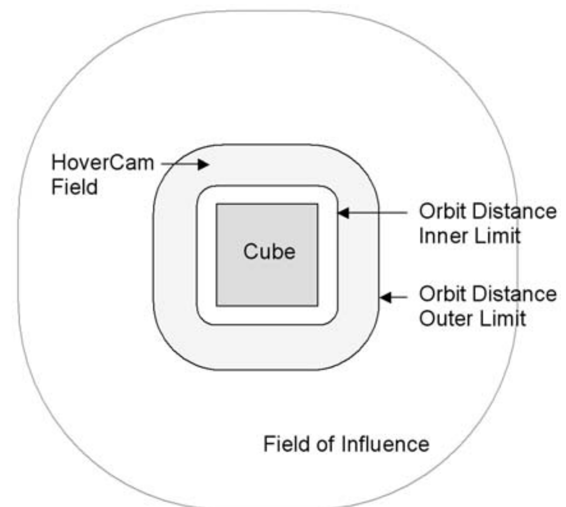


Figure 5. HoverCam Layers: A large outer shell acts a type of gravity field that interpolates traditional camera motion with the HoverCam camera motion until the Outer Limit of the Orbit Distance is reached.

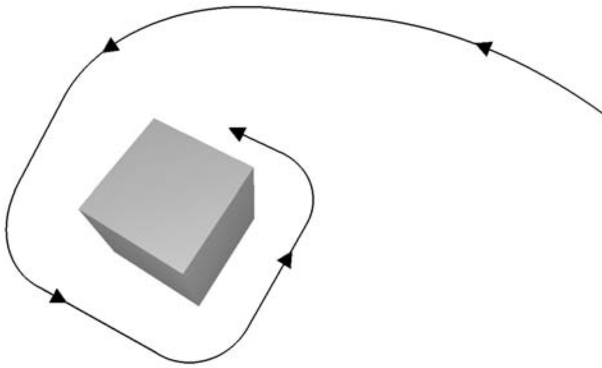

Figure 6. HoverCam Layers: As the camera approaches an object, HoverCam is slowly engaged.

### 2.3 Different Notions of "Up"

When closely inspecting an object in an abstract empty virtual environment, the problem of correctly orienting the camera, so objects do not appear to be sideways or upside-down, is not trivial. Furthermore, the model chosen to derive the up-vector at a given camera position, or given a certain camera motion, may alter the overall camera metaphor. We define four up-models: Global, Local, Driving, and Custom.

*Global*: Consider a globe representing the earth. Regardless of where the camera may be positioned, or how it moves, *up* is typically the direction toward the North Pole. For example, whether the user is looking at Australia or Sweden, the camera would be oriented so that the North Pole would be toward the top of the screen. If the user moved across the North Pole from Canada to Russia, the camera would effectively spin about 180˚ so that it would come down on the Russian side, but with the North Pole still toward the top of the screen. This constant up-vector high above the center of the scene defines our Global Up-Vector Model as shown in Figure 7.

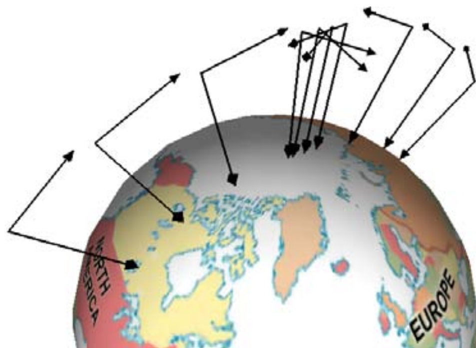


Figure 7. Global Up-Vector Model.

*Local*: In this egocentric model, the up-vector is view dependent and always points toward the top of the viewport. Therefore, moving the cursor left or right does not affect the up-vector. However, moving up or down causes the up-vector to be corrected so that the user never feels as though they have turned. For example, when moving over the North Pole of a globe from Canada to Russia, if Canada initially looked the right way up, Russia would appear upside-down. See Figure 8.

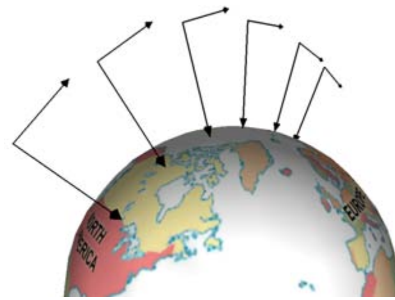

Figure 8. Local Up-Vector Model.

*Driving*: For some objects, the user may wish to have the feeling that moving the input device left or right should turn the object so that moving (the device) *up* is always "forward". Again, using the globe as an example, if we started over Brazil with the equator horizontal across the view and we moved the input device to the right, the horizon would rotate in the view until vertical, with the North Pole toward the left hand side of the screen. See Figure 9. This model could also be considered for a "flying" camera metaphor since it smoothly banks the camera in the left or right direction of mouse motion.

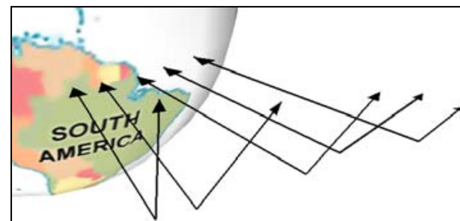


Figure 9. Driving Up-Vector Model.

*Custom*: Finally, some objects may require custom up-vector fields. For example, a model of an automobile would normally have the up-vector point from the car to high above the top of the roof. However, if a user was looking underneath the car or above the car, it may seem proper to have *up* be towards the hood. In this case, custom up-vectors could be placed on the sides of an enclosing cube, which would be interpolated based on the current camera position, to determine the current up-vector. In our current implementation, the user can move to any point in space and press a hotkey to generate an up-vector at the current position and orientation. In this way, a complex up-vector field may be authored. See Figure 10.

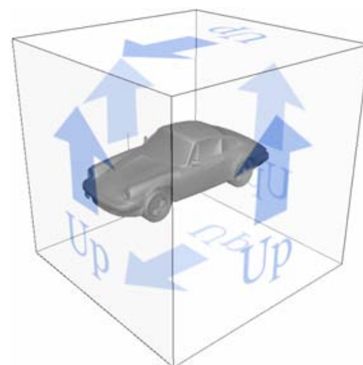


Figure 10. Custom Up-Vector Model.

### 2.4 Input Mapping

The mapping of mouse motion to camera motion may either be *push* (egocentric) or *pull* (exocentric). In a push mapping, the user conceptually pushes the camera so that, for example, moving the mouse from left to right would push the camera to the right causing the model to appear to move left. With a pull mapping, the user pulls the world in the direction of motion so that dragging from left to right moves the object from left to right, achieved by moving the camera to the left.

The name of our technique –HoverCam– implies that the user is controlling a craft floating above the surface of an object and so one may expect the push mapping to be most effective. However, given our object inspection metaphor, we typically expect to be very close to the object so that it fills most of the display. As such, when the user clicks to drag the mouse, the cursor will typically be over the object when the mouse button is clicked. This strongly conveys a metaphor of grabbing the object at that position and dragging it in the mouse direction, which implies that the camera will move in the opposite direction. For this reason, we chose the pull mapping.

Still, during a single click-drag-release input event series, a discrepancy can occur between the direction that the input device is moving and the intended camera motion in the scene. For example, for the camera motion shown in Figure 7, the user would move the mouse down until they reached the North Pole, but continuing to move down would do nothing. To move down the other side, the user would have to move the mouse in an upward direction. This can make the user feel as though they are "stuck" and this can be fairly confusing. To fix this discrepancy, HoverCam uses two up-models: one for internal calculations and one for display to the user. Internally, the Local up-model is used, which will move continuously across the top of the globe in a single drag motion without getting stuck, as shown in Figure 8. However, the *up* effect that the user sees may be any one of the four methods described above. This is implemented by applying the Local model to the camera position and orientation, followed by the application of the chosen up-model. As this update is applied during every mouse-move event, the user only feels the effect of the chosen up-model.

An appropriate choice for the up-vector model may be highly content dependent and may be a user preference or may be uniquely associated with each model in a scene.

### 2.5 Fighting Cavities

This basic algorithm shown in Figure 2 nicely handles simple convex surfaces, slightly concave surfaces, and jumps across gaps or holes. However, the true closest point may be outside the current field of view (FOV) or may even be behind the camera. In these cases, turning the camera to immediately face the new closest point would be quite disorienting and may result in some undesirable effects. This can occur if the object has protrusions, or cavities. When gliding over a cavity, for example, the closest point will jump from one edge of the cavity to the other. Step (e) of the algorithm clips the final distance traveled (of both the eye and the look-at point) to minimize these effects, slowly turning the camera to the intended position.

Specifically, to maintain smooth camera motion, Step (e) looks at the vectors (see Figure 11) from the old closest point to the new closest point ($L_0L_2$) and from the old eye position to the new eye position ($E_0E_2$). We then clip these vectors to the length $\delta$ of the input vector $i$ generated by the mouse move. This creates the final *eye* to *look-at* vector $E_3L_3$.

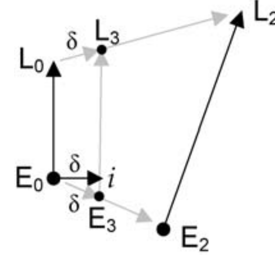


Figure 11. Motion Clipping. The final position of $E_0$ and $L_0$ are clipped from $E_2$ and $L_2$ to $E_3$ and $L_3$.

This motion-clipping step handles sharp camera turns and jumps across holes in an object or jumps across gaps to other objects. Figure 12 shows the HoverCam camera path while moving across the top of a torus (from left to right). Note how extra steps are generated across the hole in the torus, when the closest point is on the right-hand side, to smoothly turn toward the other side of the torus to continue around it.

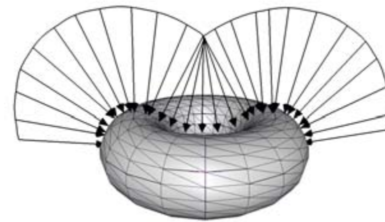

Figure 12. When moving across the hole in the torus (from left to right), HoverCam generates the extra steps needed to maintain smooth motion.

However, the camera motion shown in Figure 12 is only possible with an additional constraint. At the point where the camera is directly over the center of the torus, there are an infinite number of solutions when searching for the closest point. To resolve cases such as this, and to favor the user input, a restricted FOV constraint is added in step (b) to only look for the closest point in the general direction of the input vector (see Figure 13). There are two inputs to this constraint: the input vector $i$ and the angle of the field of view $\beta$. In our current implementation, $\beta$ is fixed at 45˚, but could be based on the view frustum. $P_0$ is the point along the vector formed by adding $i$ to $L_0$ and ensuring $L_0E_0P_0$ is ½ $\beta$. All geometry outside the triangular wedge determined by $E_0L_0P_0$, with a thickness of $2\delta$ and a length extending infinitely away from $E_0$, is disregarded during the search for the closest point to $E_0$.

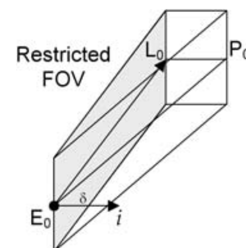


Figure 13. Restricted Search FOV. By restricting the search volume for a new closest point, the camera motion favors the user input.

This constraint helps HoverCam to handle a number of situations. In the torus example, a new closest point is found directly across the hole and the motion clipping turns the camera toward it. In concave shapes, where again, an infinite number of points are all equally close to the eye position, the user input helps to uniquely select a subset of results (see Figure 14).

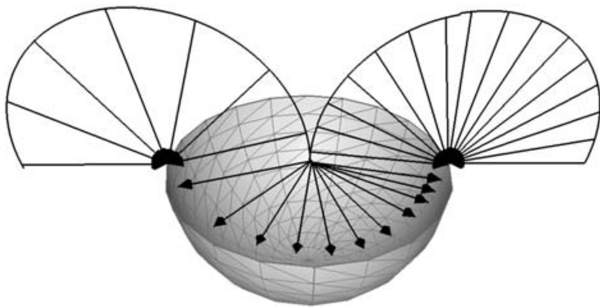

Figure 14. When moving across the inside of an open sphere (from left to right), the Restricted Search FOV and the motion clipping work together to create the expected camera motion.

To summarize, the restricted FOV for searching (in step (b)) taken together with the motion clipping (step (e)), handle the cases where there are multiple solutions thereby providing the expected camera motion.

### 2.6 Handling Sharp Turns

While the basic camera update steps outlined above generate smooth camera motion paths, tight corners can create hooks in the path that could be avoided. The problem is caused by the restricted search FOV that prevents the algorithm from finding an upcoming corner. For example, when moving right along a wall towards a corner, HoverCam looks directly ahead at the wall while panning right. However, the restricted FOV prevents HoverCam from seeing the approaching corner. The corner will eventually be found but this will push the camera back to the fixed distance from the surface effectively generating a hook in the camera path (see Figure 15).

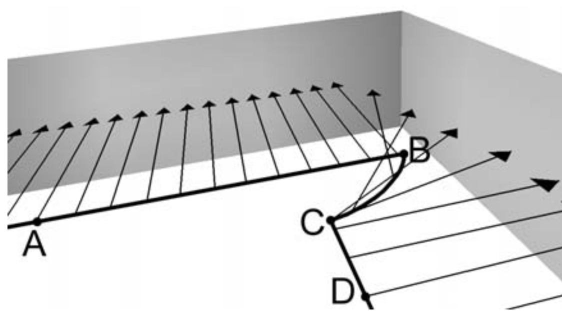


Figure 15. Hook in camera motion path when turning in a corner while moving left to right (from A to D).

To achieve the preferred trajectory, HoverCam includes a second FOV that searches for obstacles in the direction of motion (see Figure 16). The search for the closest point then includes both the restricted search FOV and the obstacle FOV. The closest point in either FOV will be considered to be the target point that we would like to veer towards.

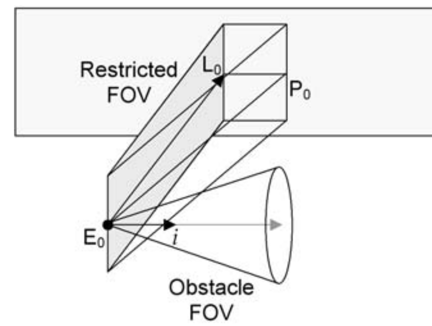


Figure 16. Restricted FOV along underlying surface and Obstacle FOV looking ahead in the direction of movement, as specified by the input vector *i*.

Now, when a corner is reached, the camera correctly turns in the direction of the input until it continues along the next wall (see Figure 17). The imminent collision with the wall is detected and the closest point will then be contained on that wall. Several steps are made while the camera turns toward it after which the camera carries on normally.

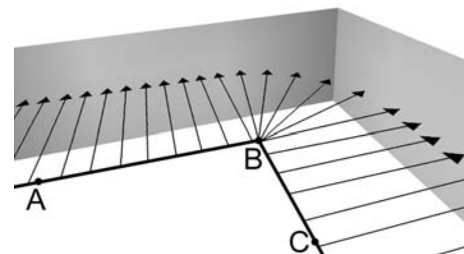


Figure 17. No hook in camera motion path when turning in a corner, while moving left to right (from A to C).

## 3 Implementation

We implemented a HoverCam prototype application in C++ under Windows XP. We added basic functionality for loading Wavefront (obj) models and rendering them using of the OpenGL graphics library. We also added visualization functions to record user input and draw the motion paths shown in the figures in this paper.

As outlined above, the general HoverCam algorithm is based on a closest point search across a polygon mesh. For obvious reasons, the naïve approach of iterating through every polygon in the model for closest point analysis would be too costly to be used for an interactive operation such as HoverCam. We therefore generate an indexing structure called a sphere-tree when the user loads an object. The sphere-tree is a hierarchal structure that encloses the polygons within our model (see Figure 18) and is built using a modified octree algorithm.

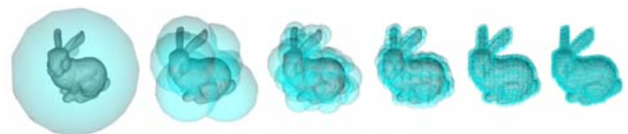

Figure 18. Six successive levels of a sphere-tree enclosing a 3D bunny model. Notice how closely the sphere-tree represents the model at the low levels.

To compute an approximate closest point on the surface of the mesh, we perform a top-down traversal of our sphere-tree, maintaining a list of all the spheres that satisfy our FOV constraints. If a sphere fails to be within either of our FOVs, we eliminate it from the list without exploring any of its children. In this manner, we eliminate a large majority of our polygons from the closest point analysis. As a further optimization, we also find the distance from the query point to the far end of each sphere, and eliminate all the spheres further than the current minimum distance. The traversal terminates when the bottom of the sphere-tree is reached, and the result is a list of the smallest spheres from the lowest level. The polygons contained within these spheres are then subjected to regular closest point analysis.

It is often the case that some of the polygons lie on the boundary of our restricted FOV. For these polygons, if the true closest point is outside the FOV, then the closest point we are interested in will lie along the intersection of our FOV with the polygon. When this happens, we perform a ray-casting technique about the perimeter of the FOV, finding the closest intersection of a ray with the polygon. If our search method ignored these cases, HoverCam would mistakenly only select closest points on polygons completely within the FOV.

## 4 Limitations

Our HoverCam algorithm essentially handles all static 3D models. The model can have significant protrusions and cavities (convex and concave areas) and may even be interior spaces such as a game level. However, moving objects may not always be handled properly, especially if moving faster than the camera. Also, models with very fine protrusions around the camera may not be found if they fall between the two FOVs being used. Any of these conditions may cause HoverCam to move inside the object or to miss it entirely.

Another limitation exists in the closest point search method. If a fast search method cannot be provided to HoverCam, interactive rates will suffer. For example, the automobile model in Figure 10 has 21,000 polygons unevenly distributed in space. Due to the high concentration of thousands of polygons in the wheels of the car, gliding across the wheels noticeably slows camera movement, despite a fairly efficient sphere tree implementation.

## 5 Initial Impressions

We showed HoverCam to six target users who were advanced 3D modelers and animators to get their initial impression. After describing the basic interaction model, we asked them to use HoverCam to inspect one of our 3D car models. All of them understood the concept and interaction mechanisms and could easily inspect the car. The camera orientation (including up-model) worked exceptionally well. In addition, a few of the users opted to use both the HoverCam and at times the traditional camera controls to inspect the car. Our system seamlessly blended the two camera styles. Finally, one user commented that HoverCam is "better than shifting between individual modes."

The most distracting usability issue appears to be the "shakiness" of the HoverCam technique as many users commented on the problem. This is an artifact of following facetted surfaces too closely. We can easily address this by smoothing normals or smoothing the model mesh. For future work, we may add level of detail support so that when HoverCam is further from the object, a smoother version of the model can be used to control the camera. In addition, two of the users requested the ability to get to an exactly framed spot (e.g., a close-up of a side mirror). The orbit distance inner limit must be small enough to allow for these types of close-up shots as we learned that our initial inner limit distance was too large. In the end, all of the users saw the value of HoverCam.

## 6 Related Work

A great deal of prior research has explored camera techniques for 3D virtual environments. Many of the techniques use 2D input from a mouse or stylus and introduce metaphors to assist the user. The most pervasive metaphor is the cinematic camera model, enabling users to rotate, pan and zoom the viewpoint. Researchers have also explored other camera metaphors including orbiting and flying [Tan et al. 2001], using constraints [Mackinlay et al. 1990; Smith et al. 2001], drawing a path [Igarashi et al. 1998], through-the-lens control [Gliecher and Witkin 1992], points and areas of interests [Jul and Furnas 1998], two-handed techniques [Balakrishnan and Kurtenbach 1999; Zeleznik et al. 1997], and combinations of techniques [Steed 1997; Zeleznik and Forsberg 1999]. Bowman et. al. present taxonomies and evaluations of various interactions and camera models [1997; 1999].

Systems that utilize higher degree-of-freedom input devices offer additional control and alternative metaphors have been investigated, including flying [Chapman and Ware 1992; Ware and Fleet 1997], eyeball-in-hand [Ware and Osborne 1990], and worlds in miniature [Stoakley et al. 1995]. Other techniques involve automatic framing of the areas of interest as typically found in game console based adventure games which use a “chase airplane” metaphor for a third person perspective. Rules can also be defined, for cameras to automatically frame a scene, that follow cinematic principles such as keeping the virtual actors visible in the scene; or following the lead actor [He et al. 1996]. Researchers have also investigated so-called *guided tours* where camera paths are procedurally determined or pre-specified for the end user to travel along. Galyean [1995] proposes a “river analogy” where a user, on a metaphorical boat, can deviate somewhat from the river, by steering using a conceptual “rudder”. Hanson and Wernert [1997; 1999] propose “virtual sidewalks” which combine virtual surfaces and specific gaze direction, and vistas along the sidewalk. Wan et al. determine a best path for automatic fly-through medical applications [2001].

The most directly related work is the UniCam [Zeleznik and Forsberg 1999] click-to-focus feature and the Tan et al. [2001] navigation system. Both of these systems are suites of camera manipulation tools and both have one feature that examines the in-scene geometry. Once the user has clicked on an object of interest, a camera path is generated to move and orient the camera toward the selected target point. The UniCam system animates the view to the new position while the Tan system uses keyboard keys to move along the generated path.

Our technique differs from these in that an updated position and orientation is interactively generated so the user is continuously in control of the camera motion and can change directions at any time. Also, the two systems mentioned do not perform collision detection or obstruction detection and so, may pass through

other polygons. Finally, HoverCam handles convex and concave shapes and models an up-vector field, whereas neither of the other systems address these aspects of navigation.

## 7 Other Applications: Volumetric Operations

In addition to surface based navigation, HoverCam can be used to intelligently view geometry without well-defined surfaces such as curves in space, point cloud data sets, or volumetric densities. To support navigation about lines or points, only the closest point search function must be changed. Figure 19 shows a HoverCam camera path made by a user moving around a set of randomly generated points (drawn as small spheres). The displayed camera path shows that HoverCam keeps the cloud data as the center of interest as the user moves around the cloud from right to left.

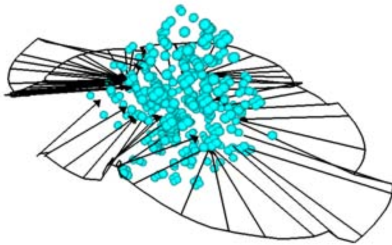

Figure 19. HoverCam Navigation about a Point Cloud.

The HoverCam algorithm can also be used to create volumetric densities. With an additional button, HoverCam can perform other operations such as selection or painting. Figure 20 shows a curve in space around which HoverCam can travel. When the user presses a modifier key, HoverCam leaves a paint trail as it moves about the curve. By increasing or decreasing the orbit distance, the user can paint closer or further from the base curve.

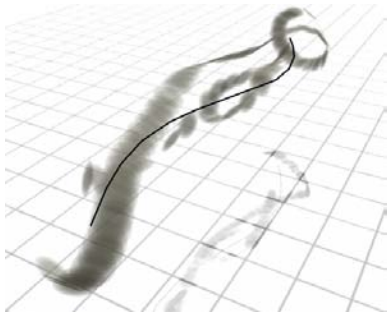

Figure 20. Painting volumetric density with HoverCam.

## 8 Conclusion

In this paper we introduced a new technique for interactive object inspection called HoverCam. The fundamental principle is to move the camera, under a small set of constraints including collision detection in the hover direction and the motion direction, followed by a small number of corrections, to maintain the hover distance from the object.

There are a number of applications of this algorithm including object inspection, volumetric operations, and interior navigation. The primary benefit to users is a simplified interaction that only requires 2D input, which can be engaged with just one button or control. Also, for object inspection, novice users can move around an object without moving to awkward positions or orientations.


## Acknowledgments

The authors thank the participants of our user study, and Don Almeida for early implementation work.


## References


BALAKRISHNAN, R. and KURTENBACH, G. 1999. Exploring bimanual camera control and object manipulation in 3D graphics interfaces. *ACM CHI*. 56-63.

BOWMAN, D., JOHNSON, D. and HODGES, L. 1997. Travel in immersive virtual environments. *IEEE VRAIS*. 45-52.

BOWMAN, D., JOHNSON, D. and HODGES, L. 1999. Testbed environment of virtual environment interaction. *ACM VRST*. 26-33.

CHAPMAN, D. and WARE, C. 1992. Manipulating the future: predictor based feedback for velocity control in virtual environment navigation. *ACM Symposium on Interactive 3D Graphics*. 63-66.

GALYEAN, T. 1995. Guided navigation of virtual environments. *ACM Symposium on Interactive 3D Graphics*. 103-104.

GLIECHER, M. and WITKIN, A. 1992. Through-the-lens camera control. *ACM SIGGRAPH 92*. 331-340.

HANSON, A. and WERNET, E. 1997. Constrained 3D navigation with 2D controllers. *IEEE Visulization.* 175-182.

HE, L., COHEN, M. and SALESIN, D. 1996. The virtual cinematographer: a paradigm for automatic real-time camera control and directing. *ACM SIGGRAPH 96*. 217-224.

IGARASHI, T., KADOBAYASHI, R., MASE, K. and TANAKA, H. 1998. Path drawing for 3D walkthrough. *ACM UIST*. 173-174.

JUL, S. and FURNAS, G. 1998. Critical zones in desert fog: aids to multiscale navigation. *ACM UIST*. 97-106.

MACKINLAY, J., CARD, S. and ROBERTSON, G. 1990. Rapid controlled movement through a virtual 3D workspace. *ACM SIGGRAPH 90*. 171-176.

SMITH, G., SALZMAN, T. and STUERZLINGER, W. 2001. 3D Scene manipulation with 2D devices and constraints. *Graphics Interface*. 135-142.

STEED, A. 1997. Efficient navigation around complex virtual environments. *ACM VRST*. 173-180.

STOAKLEY, R., CONWAY, M. and PAUSCH, R. 1995. Virtual reality on a WIM: Interactive worlds in miniature. *ACM CHI*. 265-272.

TAN, D., ROBERTSON, G. and CZERWINSKI, M. 2001. Exploring 3D navigation: combining speed-coupled flying with orbiting. *ACM CHI*. 418-425.

WAN, M., DACHILLE, F. and KAUFMAN, A. 2001. Distance-Field Based Skeletons for Virtual Navigation. *IEEE Visualization 2001*. 239-245.

WARE, C. and FLEET, D. 1997. Context sensitve flying interface. *ACM Symposium on Interactive 3D Graphics*. 127-130.

WARE, C. and OSBORNE, S. 1990. Exploration and virtual camera control in virtual three dimensional environments. *ACM Symposium on Interactive 3D Graphics*. 175-183.

WERNERT, E. and HANSON, A. 1999. A framework for assisted exploration with collaboration. *IEEE Visualization*. 241-248.

ZELEZNIK, R. and FORSBERG, A. 1999. UniCam - 2D Gestural Camera Controls for 3D Environments. *ACM Symposium on Interactive 3D Graphics*. 169-173.

ZELEZNIK, R., FORSBERG, A. and STRAUSS, P. 1997. Two pointer input for 3D interaction. *ACM Symposium on Interactive 3D Graphics*. 115-120.