Computational Aesthetics in Graphics, Visualization, and Imaging (2009)
O. Deussen and P. Hall (Editors)

CubeCam: A screen-space camera manipulation tool

Nisha Sudarsanam,[†] Cindy Grimm, [‡] Karan Singh [§]

Abstract

We present CubeCam, an image-space camera manipulation widget that uses a projected cube to both visualize the relationship of the camera to the scene and as an interaction tool to change that camera. The cube geometry reflects the use of perspective lines by artists in order to establish the scene projection. We allow the user to interactively change the camera by changing the cube's projection in the image plane. We incorporate pie menus, ghosting, and a crossing-style interface to reduce mouse movement and make it simpler for novice users to learn and experiment with the interface. Finally, we provide a novel method for visualizing camera bookmarks.

Keywords: Camera control, Projection, Perspective

1. Introduction

Manipulating the camera in a complex environment is a challenging problem. At the very least there are six degrees of freedom (position and orientation) that need to be specified, and if the full camera model is utilized this can be as high as eleven (depth of field, center of projection, aspect ratio and skew). All of these must be mapped to the two degrees of freedom supplied by a mouse or other pointer device. The traditional approach uses menu options, sliders, shortcut keys, and key modifiers to change what the mouse motion maps to. For some camera motions, such as spinning the camera around an object [Hul90] or navigating a character in a game, there is an obvious mapping from mouse motion to a restricted set of camera movements, for example, orientation or 2D position on a plane. For more general camera placement this approach begins to break down, both because the user must learn a large number of modes and also because the mouse motions may no longer map naturally to the changes in projection.

To address this problem we turn to perspective lines, a technique commonly used by artists to "sketch out" the perspective of a scene. The basic idea behind this approach is that rectilinear geometry, such as buildings or lines of trees, produces lines in the image plane that converge to vanishing points. By sketching out these lines the artist can quickly experiment with different perspective views of the scene.

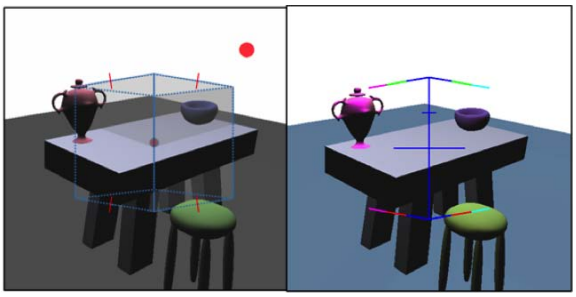


Figure 1: *CubeCam in 2-pt perspective (left) versus the IBar (right). The transparent planes provide feedback on the current focus center and a clearer picture of the cube shape.*

Previous work on camera placement, the IBar [SGS04], used a very simplified version of this to control the camera (see Figure 1). The edge of a cube is rendered in two-point perspective, and the user changes the camera by grabbing and moving different parts of the IBar. This approach is particularly effective for camera panning and zooming, but less so for other camera changes. It also suffers from the mode problem — the user has to memorize which parts of the IBar do what.

In this paper we move to a 2D widget that more explicitly captures the relationship between changing perspective lines and changing the scene projection. The user is presented

[†] e-mail: nisha.sudarsanam@gmail.com
[‡] email:cmg@cse.wustl.edu
[§] email:karan@dgp.toronto.edu

DOI: 10.2312/COMPAESTH/COMPAESTH09/065-071

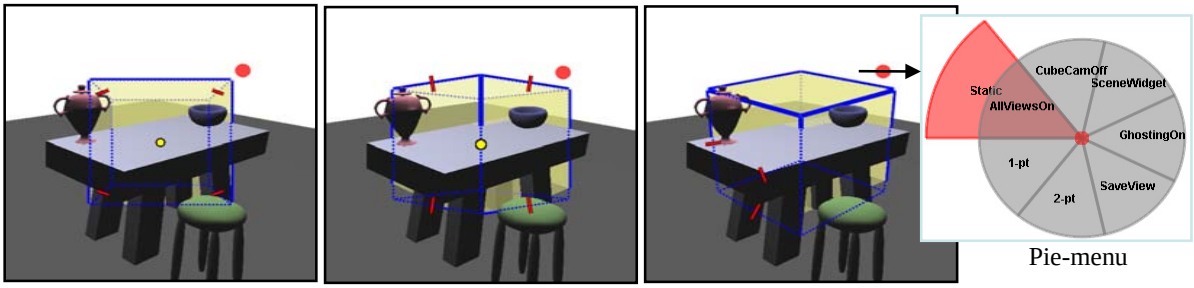


Figure 2: *The three perspective views of a cube form CubeCam. Each primitive is associated with a set of camera operations, listed underneath it. The red icon in the top right corner is the pie-menu button which when crossed pops up a pie-menu of options shown in the box.*

with a rendering of a cube and changes that 2D rendering by grabbing and manipulating parts of the cube. We use three different perspective renderings of the cube (see Figure 2) in order to ensure that there is a better cognitive match between changing the cube and the resulting perspective change. For example, a cube in three-point perspective maps naturally to rotation motions while a cube in two-point perspective is better for presenting center of projection changes.

We use a combination of Crossing [AG04] and Pie Menus [CHWS88] to minimize modes and to keep the interaction entirely in the image plane. We give the user explicit control over focus planes and rotation points through manipulation of transparent planes attached to the cube and by attaching the cube to elements in the scene. Finally, we employ ghosting both as a training device for novice users and as a visualization technique for exploring camera changes.

Bookmarks: Most applications include some method for "bookmarking" useful viewpoints. We present a novel approach to displaying some (or all) of the bookmarks in the image plane itself as part of the interaction widget. The bookmarks can be sorted by relevance to the current viewpoint, and the user can easily "page" through them to find the one they want.

Contributions: We present an in-screen, interactive widget for controlling a camera. It is particularly suited for exploring and then fine-tuning the projection.

1.1. Contributions

CubeCam is a simple, intuitive camera manipulation interface. CubeCam can visualize both the relationship of the camera with the scene and the state of the camera itself, while staying in the 2D image plane. Specifically, CubeCam explicitly visualizes important aspects of any camera manipulation interface such as the camera's focal plane and pivot point about which camera rotations take place. CubeCam provides an intuitive interface because it allows users to define camera operations in terms of the change they want to see in the projected image.

Visual aids such as ghosting of the scene help novice users learn the different functions of CubeCam and also allows advanced users to experiment with possible camera changes before actually making them. Finally, CubeCam allows users to find and visualize nearby camera bookmarks.

Overview: The paper has the following structure: Section 2 places this paper in context with previous work in this area. Section 3 describes the features of the interface including ghosting. Section 4 describes camera bookmarks while Section 6 provides the conclusion.

2. Related work

For mouse-based systems, camera control paradigms fall roughly into two categories, camera-centric and object-centric. In the camera-centric paradigm, operations are applied to the camera as if it were a real object in the scene. This mirrors camera placement in the real world, and many of the camera operations (dolly, pan, and roll) reflect that. The external parameters, position and orientation, can be specified either "through the lens", or by manipulating a pictorial representation of the camera in a second window. The internal camera parameters, with the exception of focal length, are changed through textual input.

In the object-centric paradigm, the camera is centered on an object and the viewpoint is rotated relative to the object (as if there were a virtual trackball around the object [Hul90, HSH04, KMF*08]). The camera can also be zoomed in and out. This paradigm is useful when there is a single object in the scene (or one object of importance) and the user is simply choosing a direction from which to view it.

A recent study [KMF*08] looked at using a cube for both

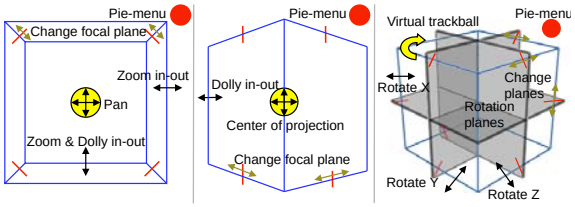


Figure 3: *Diagrams of the three views. Any of the edges work, i.e., all of the outside edges in 1-point perspective will change the zoom.*

control and visualization of the view direction. They compared clicking on the cube, menu selection, and trackball when trying to match a given view. Interestingly, trackball was both the fastest and the most preferred by the users. The authors suggest that interactivity is key here — even if the user moves in the wrong direction, they can quickly correct.

Three or six degrees of freedom devices permit other interesting navigation techniques [BKH97], such as the palmtop world [SCP95], the "grab and pull" approach [PBWI96] and virtual fly-throughs [WG95]. The latter can also be used in mouse or keyboard-based systems if the camera's movement is restricted to a well-defined floor plane (most first-person shooters use this approach).

An alternative approach to directly specifying the camera is to use image-space constraints [Bli88, GW92]. In this approach, points in the scene are constrained to appear at particular locations, or to move in a specified direction, and the system solves for the camera parameters that meet those constraints.

The recently-introduced IBar [SGS04] is, in some sense, a specialization of the constraint approach, where the points are the points of the edge of a cube. Like CubeCam, the IBar is a screen-space widget where changing the widget changes one or two camera parameters. The IBar and CubeCam have similar goals; both systems move beyond current menu-based camera manipulation techniques to a unified screen-space camera primitive. The underlying mathematical framework of the two systems are similar. Thus, both systems support the same set of camera operations. However, CubeCam improves and extends the interface presented to the user.

- **(improved) Rendering:** The IBar's shape provides information on the perspective, but has no feed back on rotation or focus points.
- **(improved) Camera primitive Interface:** In the IBar interface, each camera operation is associated with one part of that widget. Some of these associations are not obvious and tend to be confusing for a novice user. The CubeCam associations are more natural.
- **(improved) Rotation:** In addition to rotation around the focus point, CubeCam also supports rotation around an arbitrary point in the scene or around a selected axis.
- **New Features:** CubeCam supports both visualizing and manipulating the focal plane and the rotation point. Ghosting is used to preview camera changes. Finally, CubeCam supports visualization and classification of camera book marks in-screen.

3. The CubeCam Interface

We first discuss the interface elements that are common to all of the perspective views. We then describe, for each perspective view, which camera operations are available and how the user invokes them.

For all mouse-widget selection interactions we employ a variation of the CrossY style [AG04] approach, which lets the user perform selection with strokes instead of multiple clicks. For example, there are several actions which affect the camera widget but are not camera operations themselves, namely: switching between perspective views, switching between object- and scene-centric, toggling ghosting, toggling bookmarks, changing bookmark placement algorithms, and hiding the CubeCam interface. Rather than use a traditional menu or check boxes, we use a pie-menu which is invoked by crossing a dot on the screen (see Figure 2). This allows the user to bring up the pie-menu, select their choice, and close the pie-menu all in one stroke.

All three perspective views allow the user to interactively specify the focal distance. This is not a camera parameter per-se, but it does change the effective behavior of certain camera motions. For a zoom plus a dolly-in or a center of projection change, the focal distance specifies what part of the scene will stay the same size. For rotations, it is the point around which the rotation occurs. To select the focus distance the user slides a transparent plane through the scene using the red handles on the widget (see Figure 3). To select a rotation point the user positions three planes — this allows the user to pick *any* point in the scene, not just a point along the view vector, which is what most systems support.

The CubeCam widget can operate in either scene-centric (centered on the 2D screen) or object-centric (centered on the object's projected 2D bounding box). Note: If the widget's controls extend off the screen we scale the widget down so that they are still reachable. In scene-centric mode, the CubeCam always snaps back to the center of the screen and to a default orientation after the camera manipulation is complete. This is useful for navigating around the scene as a whole. In object-centric mode, the CubeCam widget is tied to the bounding box of the currently selected object. This is useful for positioning and orienting a specific object at a particular location in the 2D image plane.

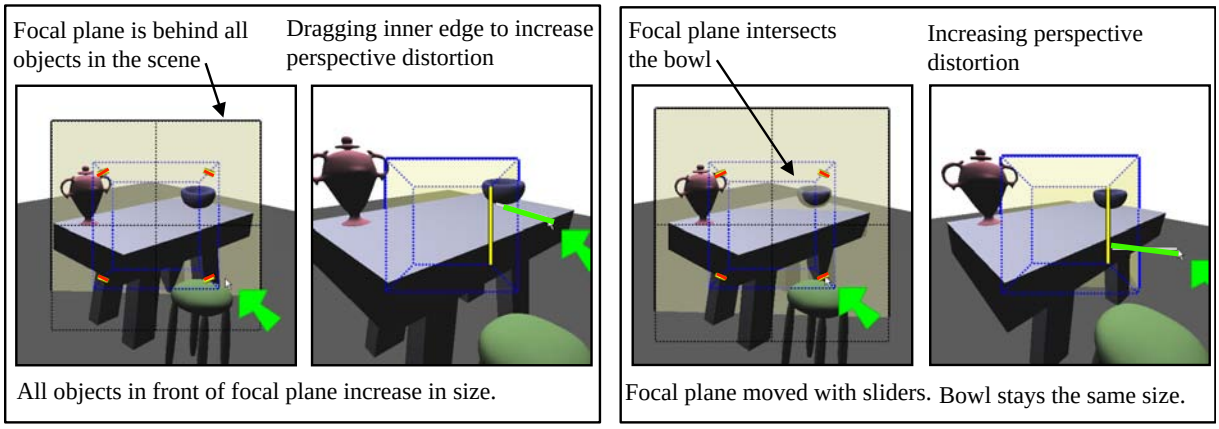


Figure 4: *Using the position of the focal plane to control the effects of perspective distortion.*

3.1. 1-Point Perspective view

The 1-pt perspective view supports zooming (by making the entire projected cube bigger or smaller), panning (by moving the projected cube), and perspective distortion (by changing the relative projected sizes of the front and back of the cube).

The degree of perceived perspective distortion is a function of the distance of the camera eye point to the object in question — the closer the camera is to the object, the more distortion. Unfortunately, changing the camera distance also changes the projected size of the objects in the scene. To counter-act this, the system automatically adjusts the zoom to keep objects at the specified focal distance the same size [GS05]. By changing the focal distance using the focal plane, the user can control what part of the scene remains fixed (see Figure 4).

3.2. 2-Point Perspective View

2-pt perspective view supports camera dollying in and out and center of projection change. A camera dolly is specified by making the cube bigger or smaller (see Figure 5). To change the center of projection, move the cube's centerline. Changing the center of projection causes the scene to "slide" in the opposite direction. To counter-act this, the system automatically pans the camera in the opposite direction [GS05], keeping objects on the focal plane in the same spot. The user can select what stays still by positioning the focal plane using sliders on the slanting edges of the cube (see Figure 6).

3.3. 3-Point Perspective View

The 3-pt perspective view supports both constrained rotation around an axis or traditional virtual trackball manipulation (see Figure 7). This is a two-stroke action; the first stroke determines the type of rotation, the second actually performs

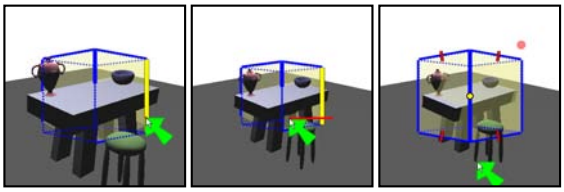


Figure 5: *Camera dolly in the 2-pt perspective view.*

the rotation. To initiate trackball rotation, the user crosses a corner of the cube (two edges). To perform a constrained rotation the user crosses a single edge (the one corresponding to the desired rotation axis). Now the user clicks and drags on any point on the cube to perform the actual rotation. Note that if the camera is in object-centric mode then the available rotation axes are determined by the object's coordinate system.

By default, the camera will rotate about the focal point (scene-centric) or the center of the selected object (object-centric). This point can be both visualized and changed using three semi-transparent, perpendicular planes (see Figure 8). The planes are aligned along the principal axes of the cube and can be re-positioned using sliders located on the three edges. The rotation point is the intersection of the rotation planes. This provides the user with explicit control over the desired center of rotation.

3.4. Ghosting

Our goal is to present the user with an in-screen camera manipulation interface that naturally encapsulates specific projection changes. By creating three perspective views (and hence three versions of the widget) we simplify the individ-

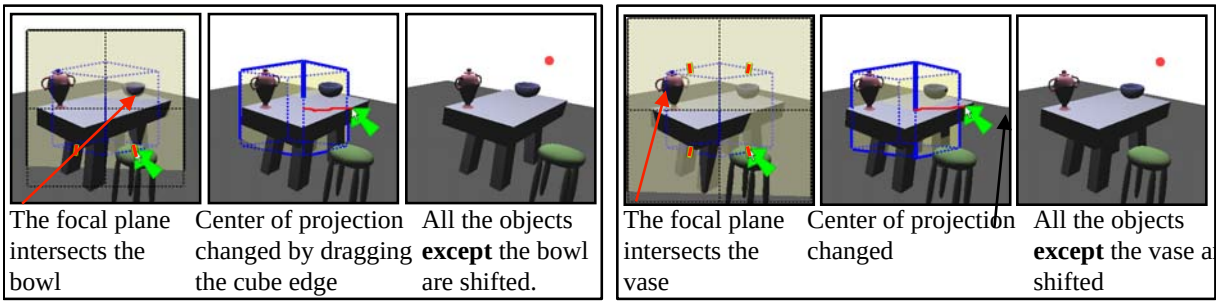


Figure 6: *Using the focal plane to determine which object stays still when changing the center of projection.*

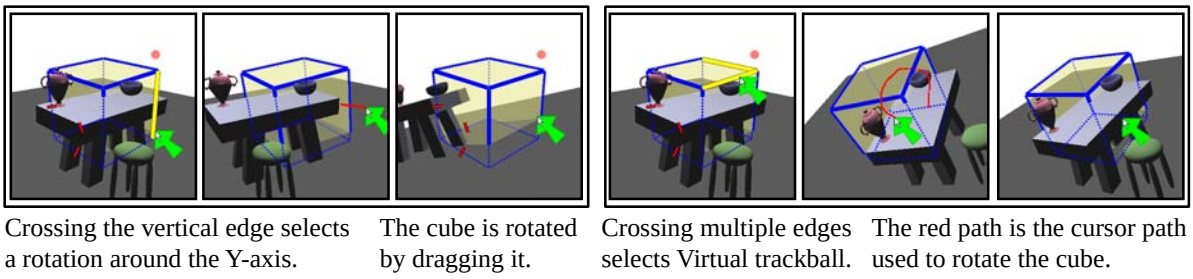


Figure 7: *Constrained rotation (left) and Virtual trackball (right).*

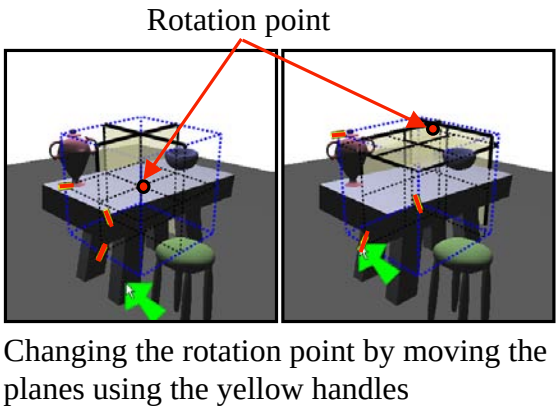


Figure 8: *Positioning rotation planes to specify the rotation center.*

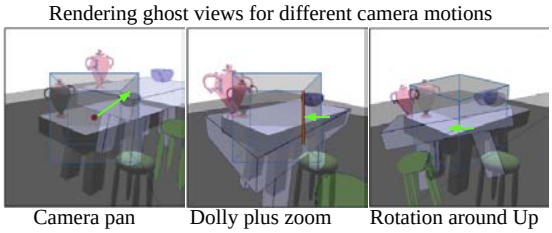


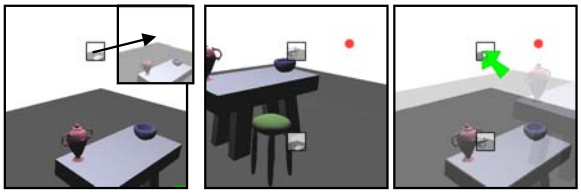
Figure 9: *Ghosting helps users see the effects of the camera movement before doing it. Green line shows direction of mouse movement.*

ual widgets but complicate the overall interface. In order to help the user interactively learn what different actions do we turn to ghosting. When ghosting is active, users manipulate the camera primitive as they would normally, but instead of actually changing the projection the result is presented as a ghosted overlay on the original scene. This allows the user to see both the original projection and the changed widget and projection in the same view (see Figure 9). This is both a learning tool and a way for advanced users to experiment with possible camera changes before actually making them.

To create a ghost camera the current scene is rendered to the back-buffer using the ghost camera. The scene is rendered under the original lighting but in a non-photorealistic style with the silhouette edges highlighted. The contents of the back-buffer are copied into a texture which is alpha-blended on top of the original scene.

4. Camera Bookmarks

It is very useful to be able to save cameras and snap back to them at will. For example, when modeling a surface, a user might bookmark a handful of orthogonal views and close-



Bookmarking two views. Most similar view is placed on top.

Mousing over icon shows ghost view.

Figure 10: *Adding two bookmarks, sorted by eye point. Icon is made by rendering the scene with silhouette edges (inset shows blowup of icon image). Right: Mousing over the icon shows a ghosted view of the bookmark.*

ups of complex geometry. An animator might also use bookmarks to start laying out an animation sequence. In both of these cases, we need to provide the user with a method for quickly searching through existing views. Although the user could simply create a text list, appropriately naming each camera, we believe that a visual search mechanism is more useful and faster. The bookmark "list" is displayed as icons arranged on the screen, with each icon showing an image of the scene from that view. As the user mouses over the icons a ghost image is rendered on top of the current scene. Double clicking on the icon switches to that bookmark's view.

We support two bookmark placement algorithms. The first is a simple static approach — the user simply places the icon where they want it on the screen. For example, for key framing an animation the user might place the icons sequentially across the top of the screen.

The second approach automatically places the bookmark icons in a circle on the screen. The bookmarks are sorted by similarity to the current view, with the most similar view at the top. If the CubeCam is in 1-point or 2-point perspective view, the ordering is based on the relative eye points of the cameras, otherwise, it is based on orientation.

To order cameras based on eye point, each bookmarked camera's eye point is projected onto the film plane of the current camera. The magnitude of the vector from the origin to the projected eye point is used to order the bookmarks (Figure 10).

For orientation, the rotation distance is calculated by measuring the length of the geodesic path between the quaternion of the current camera, q_c and the quaternion of the bookmarked camera q_i (Figure 11).

$$GeodesicPath(q_c, q_i) \;=\; Log({q_c}^{-1}, q_i)$$

As the number of bookmarks in the scene increases, the circle of bookmarks created by the automatic placement algorithm becomes more crowded. To prevent the occlusion

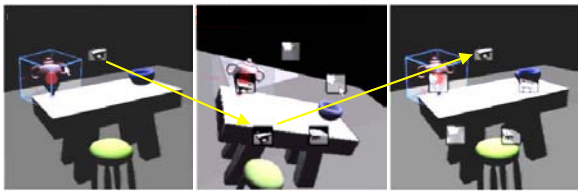


Figure 11: *Bookmarks ordered by similarity of orientation.*

of bookmarks in the circle, we vary the size of the icon as a function of the number of bookmarks in the scene by making the size $\frac{2\pi\sqrt{2}r}{3n}$, where r is the radius of the circle and n is the number of bookmarks.

Similar to ghosting, we render the scene with edges highlighted when making the bookmark icon.

5. Examples and Remarks

The accompanying video demonstrates the complete functionality of the CubeCam and also includes an example of using CubeCam to create an extreme perspective view of a table with some objects.

6. Conclusion

We have presented CubeCam, a simple intuitive screen-space camera manipulation widget. CubeCam supports visualizing and changing the camera in image-space, allowing the user to better understand the relationship of the camera to the scene. Users can explicitly visualize and adjust the camera's focal-plane and rotation point. Visual aids such as ghosting help users remember the different operations associated with each camera perspective view. Finally, we demonstrate a novel, in-screen visualization technique for camera bookmarks that incorporates current view information.

Acknowledgments: This work was funded in part by NSF grant 0238062.

References

[AG04] APITZ G., GUIMBRETIÈRE F.: Crossy: a crossing-based drawing application. In *UIST '04: Proceedings of the 17th annual ACM symposium on User interface software and technology* (New York, NY, USA, 2004), ACM, pp. 3–12.

[BKH97] BOWMAN D. A., KOLLER D., HODGES L. F.: Travel in immersive virtual environments: An evaluation of viewpoint motion control techniques. *IEEE Proceedings of VRAIS'97*, 7 (1997), 45–52.

[Bli88] BLINN J.: Where am i? what am i looking at? In *IEEE Computer Graphics and Applications* (1988), vol. 22, pp. 179–188.

[CHWS88] CALLAHAN J., HOPKINS D., WEISER M., SHNEIDERMAN B.: An empirical comparison of pie vs. linear menus.

In *CHI '88: Proceedings of the SIGCHI conference on Human factors in computing systems* (New York, NY, USA, 1988), ACM Press, pp. 95–100.

[GS05] GRIMM C., SINGH K.: Implementing the ibar camera widget. *Journal of Graphics Tools 10*, 3 (November 2005), 51–64. This is the full implementation details for the UIST 2004 paper. There is source code available.

[GW92] GLEICHER M., WITKIN A.: Through-the-lens camera control. In *Siggraph* (July 1992), Catmull E. E., (Ed.), vol. 26, pp. 331–340. ISBN 0-201-51585-7. Held in Chicago, Illinois.

[HSH04] HENRIKSEN K., SPORRING J., HORNBAEK K.: Virtual trackballs revisited. In *IEEE Transactions on Visualization and Computer Graphics* (Mar 2004), vol. 10, pp. 206–216.

[Hul90] HULTQUIST J.: A virtual trackball. In *Graphics Gems*. 1990, pp. 462–463.

[KMF*08] KHAN A., MORDATCH I., FITZMAURICE G., MATEJKA J., KURTENBACH G.: Viewcube: a 3d orientation indicator and controller. In *SI3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2008), ACM, pp. 17–25.

[PBWI96] POUPYREV I., BILLINGHURST M., WEGHORST S., ICHIKAWA T.: The go-go interaction technique: Non-linear mapping for direct manipulation in VR. In *ACM Symposium on User Interface Software and Technology* (1996), pp. 79–80.

[SCP95] STOAKLEY R., CONWAY M. J., PAUSCH R.: Virtual reality on a WIM: Interactive worlds in miniature. In *Proceedings CHI'95* (1995).

[SGS04] SINGH K., GRIMM C., SUDARSANAM N.: The ibar: A perspective-based camera widget. In *UIST* (October 2004).

[WG95] WLOKA M. M., GREENFIELD E.: The virtual tricorder: A uniform interface for virtual reality. In *ACM Symposium on User Interface Software and Technology* (1995), pp. 39–40.