Camera Control in Three Dimensions with a Two-Dimensional Input Device

Mark A. Livingston[1,2], Arthur Gregory[2], and Bruce Culbertson[1]

[1]Hewlett-Packard Laboratories
[2]Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

We implement a variety of operations to control a virtual camera using a two-dimensional modal input device, such as a standard three-button mouse. The operations offer control of position, orientation, or zoom. All are designed to be easy to use for the end user, but we also designed for ease of implementation for the programmer.

Keywords: view control, mouse, user interface, interactive graphics, 3D graphics, rotation, orientation, translation, zoom, navigation

1 Introduction

Controlling objects and virtual cameras in three dimensions challenges the user interface designer. Frequently the user must control all three position degrees of freedom (DOFs), all three orientation DOFs, and a zoom DOF with only a two-dimensional modal input device, such as a mouse. In this paper, we present a set of operations that together can implement seven DOF navigation for the virtual camera. Some of these operations duplicate each other, but since user tastes and intuitions behind applications can vary, we find that each operation has a place in our user interface toolkit. Table 1 lists the operations and the camera parameters they affect.

Operation	DOF	Description
Orbit	2	Change dir to camera from object
Roll	1	Change vertical dir on-screen
Zoom	1	Scale view volume
Magnify	1	Scale view of the world
Translate	1	Push/pull camera from/to model
Dolly	1	Translate along principal view ray
Truck	2	Translate parallel to image plane

Table 1: New user interface operations in our toolkit. Note that we have new versions of the orbit and translate operations that implement specific constraints to make the motion more intuitive for the user.

2 Previous Work

ARCBALL [4, 5] allows the user to control the orientation of the environment with respect to the camera[1]. The user directly specifies an arc on a sphere which is rigidly attached to the environment. The arc is defined by two vectors which originate at the center of the sphere. Mouse locations on the screen define the vectors via rays from the virtual camera center through the mouse location on the screen and ultimately through the sphere. One vector terminates at the point on the surface of the sphere where the rotation operation was begun (mouse click); the other terminates at the current point (mouse drag). The arc denotes a rotation of the sphere relative to the camera, as if the camera were orbiting the sphere. This removes any dependence on the path by which the mouse arrived at its current position, and allows the user to easily return to the initial view. We believe these to be desirable properties and incorporate them in our design. However, with ARCBALL, control of the "up" direction is difficult to maintain.

The UniCam interface [7] uses gestures and screen-space subdivision to provide six DOF control with a single-button mouse. Rotation operations occur with mouse motions near the borders of windows. User gestures with the mouse rather than user interface buttons (e.g. those provided on the mouse) disambiguate among the directions along which translation occurs or around which rotation occurs.

Other applications, for example many VRML browsers, map user actions to navigation operations with widgets or in other ways. What is important to note, however, is that the operations provided frequently lack in control of the sense of "up" in the world as the user navigates, similar to ARCBALL. Other systems map mouse motion, velocity, and acceleration to different navigation operations. We feel these gestural interfaces place too many requirements on the user for easy and accurate navigation.

[1]Orientation-only controllers can be thought of as translating the camera while maintaining a fixed point of observation, sometimes known as *orbiting* the object. These two operations are indistinguishable on the screen if the entire environment is affected by the operation, and thus we refer to orbiting operations as changes in the orientation, regardless of the implementation.

3 Navigation Operations

The previous work together can cover six DOF navigation for a camera. We introduce a set of operations and mappings that provide seven DOF navigation for the virtual camera.

3.1 Orbiting with Fixed Vertical

We designed an orbiting method that would strictly control the sense of "up" in the world as an alternative to ARCBALL's orbiting operation.

The geometry of the ARCBALL interface can be understood by examining the "look-at" transformation [1]. ARCBALL offered the user control of three DOF. These were understood as the orientation of the environment (or, more precisely, the orientation of a sphere rigidly affixed to the environment), but could equivalently be described as the direction of the camera (with respect to a fixed point of attention, or *look point*, and a fixed distance from that point, for two DOF) and the world direction that is vertical on the screen.

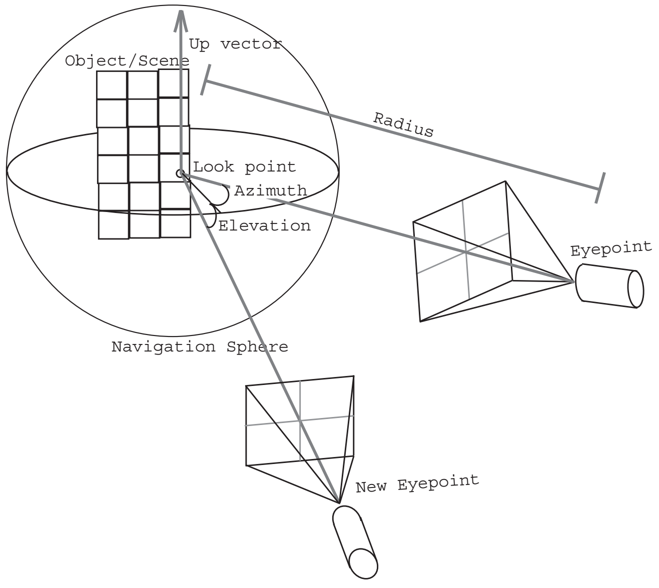

Figure 1: The orbit operation allows the user to change the azimuth and elevation of the camera while maintaining a fixed radius from the look point. The direction of the up vector as seen by the camera also remains fixed. The dolly operation will enable a change in the radius.

Orbiting changes the direction from the look point to the camera. For a fixed look point, we can map mouse motion to changes in the azimuth and elevation of the camera position with respect to the look point. This implies that during the operation, a fixed radius exists between the camera and the look point (Figure 1). This mapping is natural for a camera positioned and oriented with the **gluLookAt** call in OpenGL [6]. It also implies that we can fix the direction of the up vector in the world. Because the view can change without changing the up vector, note that the up vector will not necessarily be perpendicular to the principal view ray.

Note that since we give the user explicit control of the "up" direction in the camera roll operation, it is reasonable to prevent gimbal lock by limiting the change in the elevation angle to just shy of the poles. The user can change the up vector to enable viewpoints from those locations relative to the object.

3.2 Camera Roll

Rotation around the principal view ray remaps the direction of the up vector on the screen. This implies that the up vector is rotated in the world. One's first inclination might well be to assume that the axis for this rotation is simply the principal view ray. But as noted above, these two vectors are not necessarily perpendicular. We found that using the principal view ray as the rotation axis thus yielded unintuitive control of the rotation. Rotation in a plane parallel to the image plane is hard to map to the cursor; the user would have to move the mouse in a circular path. Instead, we shear the up vector along the direction of the camera x axis (Figure 2). Though slightly nonlinear and only approximate, the intuition of where the up direction is on-screen outweighs the slight change to the operation.

A normalization of the sheared up vector is necessary, but can be done once at the release of the operation. The

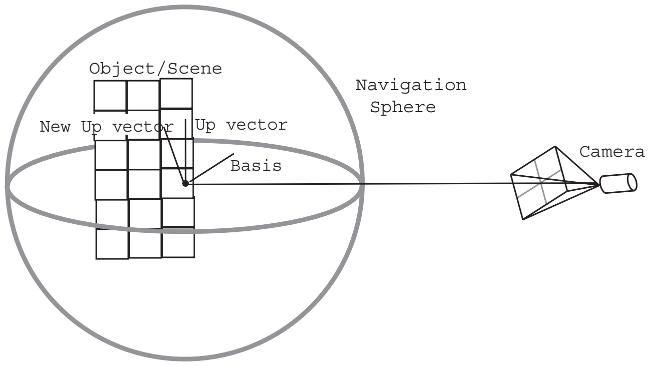

Figure 2: The roll operation allows the user to change the direction of the up vector as seen by the camera, which is a rotation of the up vector. The plane in which this rotation takes place is spanned by the current up vector and a basis vector which is parallel to the camera x axis. Note that the axis of rotation is not parallel to the camera principal axis and must be computed to perform the rotation. Our alternative implementation simply shears the up vector along the direction defined by the camera x axis.

nonlinearity could be reduced by pre-warping the shearing distance.

3.3 Zoom and Magnify

When a mouse button is pressed, the cursor location denotes a ray through the camera image plane to the object. We intersect this ray with the scene (taking the closest point) to determine the center point[2] for the magnification. We can magnify under either orthographic or perspective projection. We scale the view of the objects in the world (but not the camera position) around the intersection point. We maintain a scaling matrix (with appropriate pre- and post-translations to center the scaling) between the actual world and the displayed world.

This is a similar effect to a zoom operation, i.e. changing the focal length of a perspective camera or scaling the view volume boundaries of an orthographic camera. Some of our interfaces implementation a true zoom operation. Zoom operations are easily performed with the view volume commands provided in the graphics library [6]. Our magnify operation, however, changes the view of the world without modifying either the scene or camera parameters, which are both representations of real-world objects in some of our data sets.

An off-center zoom operation is still possible, however, regardless of the function used to define the camera model. Note that the off-center zoom can translate or rotate the view volume with respect to the camera's intrinsic coordinate system, as well as change the view volume size. These are parameters of the camera we have not expressed before, but since we parameterize them by the focal length or (in the magnify operation) by the scaling, we are not really adding new parameters to the camera model. We are, however, aliasing these extra parameters as a function of the zoom (or magnify) parameter.

[2]This point is known as the *hit point* in UniCam [7].

```
constant magnifyFactor = 0.1
magnifyVal = 1

MagnifyUp( )
    magnifyVal *= 1 + magnifyFactor

MagnifyDown( )
    magnifyVal /= 1 + magnifyFactor

...in display loop...
SetWorldToViewerTransform( )
glTranslate( - centerPoint )
glScale( magnifyVal )
glTranslate( centerPoint )
DrawWorld( )
```

Figure 3: OpenGL pseudocode of magnify operation.

3.4 Translation With Recentering

This operation allows the user to navigate the model by pushing the camera away from or pulling it towards a point on the model (defined by a ray, which is in turn defined by a mouse click), and optionally performing the same push or pull on the center of rotation. This is similar in spirit to the combination of "dollying" and "orbiting about a specific point" provided by UniCam [7].

By changing the position of the center of the orbiting operation during translation (Figure 4), we affect future orbiting operations. If the user has translated forward, part of the model may now be behind the camera. A subsequent rotation can then reveal previously hidden portions of the model. This can be a powerful tool for examining the "interior" regions of concave objects (Figure 5).

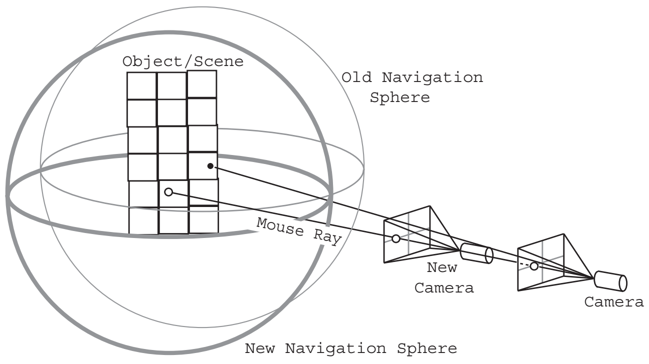

Figure 4: Translation with recentering moves the camera forward along a ray defined by the mouse position, and pushes the orbiting center towards the point at which the model first intersects that ray.

Under orthographic projection, the view generated by this translation will not change if the mouse location is in the center of the image plane and the translation is only forward– i.e. strictly a "dolly" operation. Of course, if the translation is far enough straight forward, then the camera may pass through part of the scene, clipping it from the view. Mouse locations not in the center will induce translations in the other dimensions as well.

With the finite z resolution available in the standard graphics frame buffer, we must also attenuate the farplane distance in the projection matrix (Figure 6).

```
MoveInit( )
    camRay = ImagePointToWorldRay( $x$, $y$,
        *world-to-camera-matrix* )
    newCenter = Intersect( camRay, model )

MoveIn( )
    MoveInit( )
    centerDispl = newCenter − center
    camCenter = camCenter + centerDispl * zoomFactor
    lookAt = lookAt + centerDispl * zoomFactor
    farPlaneScale *= 1 + zoomFactor

MoveOut( )
    MoveInit( )
    centerDispl = newCenter − center
    camCenter = camCenter - centerDispl * zoomFactor
    lookAt = lookAt - centerDispl * zoomFactor
    farPlaneScale /= 1 + zoomFactor
    if( farPlaneScale < 1 )
        farPlaneScale = 1
```

Figure 6: Implementation of translation with recentering of the user's view of the environment for orthographic projection. The variable `farPlaneScale` determines the far plane distance, both in orthographic or perspective projection.

Under perspective projection, we perform the same operation: translate the camera position along the specified ray and translate the center of the orbiting operation. In this case, any translation will change the view, although translation straight forward can produce similar images to zooming. It also influences future rotation operations by changing the center of rotation.

Our implementation takes advantage of the pick function in the OpenGL graphics library utilities [6]. The graphics engine can identify those polygons that currently project to a given screen point. Hence we only need to compute the intersection point with a few primitives, usually one or two. Although this works well for models that are decomposed into polygons before drawing, some data sets might not be displayed in this fashion. It would be nice to eliminate this dependence. .

3.5 Dolly

The ARCBALL orbiting mode changes the direction from the look point to the camera, but keeps the distance constant. A dolly operation changes this distance (Figure 1). This can be confounded with a zoom operation for small motions, but in fact is translating the camera forward along its principal ray.

3.6 Truck

In order to provide easier access to translation, we also implement the truck operation, which is translation constrained to the directions spanned by the image plane basis. These basis vectors are trivially obtained by examining the world-to-camera transformation matrix. We simultaneously translate the look point by the same distance as the camera (Figure 7). We find this produces a more natural motion in the image.

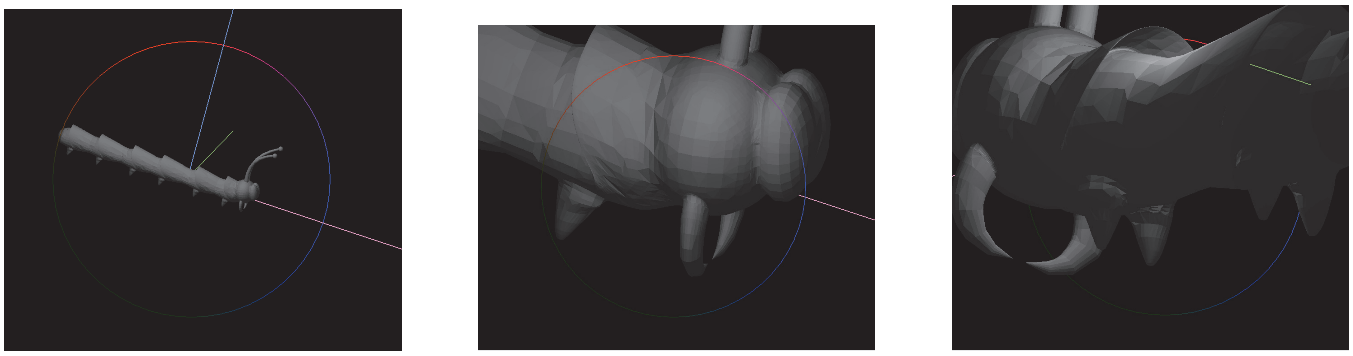

Figure 5: Demonstration of translation with recentering. The leftmost image shows the initial pose. The user then clicks and holds the mouse button on the thorax (just behind the head) to translate forward with recentering. The middle image show the result of this operation. Next the user orbits the thorax, passing through the body of the model to get a better look at the head from behind (rightmost image).

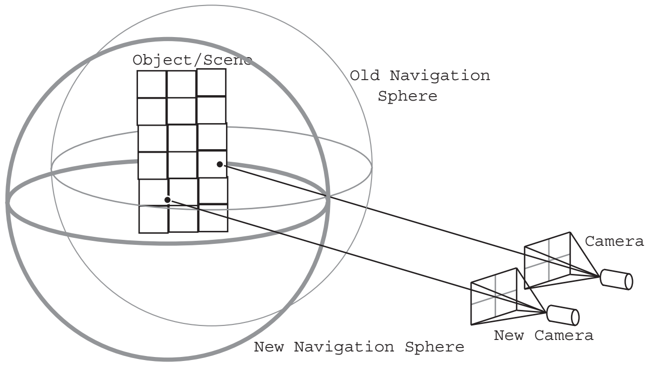

Figure 7: The truck operation translates the camera viewpoint and the look-at point parallel to the image plane.

3.7 Constrained Motions

Some operations described above affect two DOFs at a time. We can easily add constraints to the orbit and truck operations so that only one DOF is affected by the mouse motion, as was done for ARCBALL. We select the dominant direction of the mouse motion, defined by the one in which the motion first reaches a threshold of the number of pixels the mouse moves. We then eliminate any motion in the other dimension for the duration of the current operation. This adds some bookkeeping and conditional statements to the implementation.

4 Conclusions

We have found this to be a useful set of navigation operations which carry the primary advantages of the ARCBALL interface: intuitive control for the user and independence of the operations from the path the mouse takes on the screen. We prefer multiple buttons and key modifiers to engage the various operations rather than rely on the user to perform gestures on the screen, which, however small they may be, seems to retreat from ARCBALL's goal of path independence. These operations together can provide six-DOF navigation control and control of the view volume width.

One interface using some of these operations was used for a demonstration application at a conference. Users there had little, if any, trouble navigating around the environment. (Most were presumably knowledgeable of user interfaces.) This hardly substitutes for a formal study [2], but does provide encouragement that the operations are sound and successfully mimic the usability of ARCBALL and UniCam. We have had success navigating under two separate implementations of interfaces using different subsets of these operations and find them suitable to our needs.

Web information

This operations are simple to implement with vector geometry operations. Pseudocode for some can be found at

`http://www.cs.unc.edu/~livingst/navigate.html`

along with more detailed implementation notes.

References

[1] Blinn, J. Where am I? What am I looking at? *IEEE Computer Graphics and Applications 8*, 4 (July 1988), 76–81.

[2] Hinckley, K., Tullio, J., Pausch, R., Proffitt, D., and Kassell, N. Usability analysis of 3d rotation techniques. In 10th *ACM Symposium on User Interface Software & Technology (UIST'97)* (Oct. 1997), pp. 1–10.

[3] Hultquist, J. *A Virtual Trackball*. Graphics Gems I. Academic Press, 1990, pp. 462–463.

[4] Shoemake, K. ARCBALL: A user interface for specifying three-dimensional orientation using a mouse. In *Proceedings of Graphics Interface '92* (May 1992), pp. 151–156.

[5] Shoemake, K. *Arcball Rotation Control*. Graphics Gems IV. Academic Press, 1994, pp. 175–192.

[6] Woo, M., Neider, J., and Davis, T. *OpenGL Programming Guide*, 2$^{\text{nd}}$ ed. Addison Wesley Developers Press, 1997.

[7] Zeleznik, R. C., and Forsberg, A. UniCam–2D gestural camera controls for 3D environments. In *1999 ACM Symposium on Interactive 3D Graphics* (Apr. 1999), pp. 169–174.