◊ III.1
Arcball Rotation Control

Ken Shoemake
University of Pennsylvania
Philadelphia, PA
shoemake@graphics.cis.upenn.edu

◊ Introduction ◊

Previous Gems have explained how to manipulate rotations in 3D with a virtual trackball (Hultquist 1990), and in both 3D and 4D with a rolling ball (Hanson 1992). Both methods are essentially the virtual sphere of a recent survey (Chen *et al.* 1988), and simulate some physical action. In so doing, however, they exhibit *hysteresis*, or path dependence. That is, when you drag the mouse from point A to point B, the end result will change depending on the path you follow. Hanson uses this effect as a way to rotate around the axis perpendicular to the screen (which I will call z), but usually it is just a counterintuitive nuisance. This Gem presents C code for the Arcball rotation controller (Shoemake 1992), which is path independent. It is cheaper to implement than the other methods, but better behaved and more versatile. One special feature of Arcball is its ability to handle with equal ease both free rotation and constrained rotation about any axis. The simplest implementation uses quaternions (Shoemake 1985).

◊ Arcs to Rotations ◊

Recall that a unit quaternion $q = [(x, y, z), w] = [\hat{\mathbf{v}} \sin\theta, \cos\theta]$ represents a rotation by 2θ around the axis given by the unit vector $\hat{\mathbf{v}}$, and that the quaternion product qp represents the rotation p followed by q. Now suppose we have two points on a unit sphere in 3-space, $\hat{\mathbf{v}}_0$ and $\hat{\mathbf{v}}_1$, considered as unit quaternions $[\hat{\mathbf{v}}_0, 0]$ and $[\hat{\mathbf{v}}_1, 0]$. Their "ratio" $\hat{\mathbf{v}}_1\hat{\mathbf{v}}_0^{-1}$ converts the arc between them to a rotation.[1] What rotation do we get? Because the points give us pure vector quaternions, we have $\hat{\mathbf{v}}_1\hat{\mathbf{v}}_0^{-1} = [\hat{\mathbf{v}}_0 \times \hat{\mathbf{v}}_1, \hat{\mathbf{v}}_0 \cdot \hat{\mathbf{v}}_1]$. Thus the axis of rotation is perpendicular to the plane containing the two vectors, and the angle of rotation is twice the angle between them.

The Arcball controller displays this sphere on the screen (cheaply, as the circle of its silhouette), and uses the mouse down and drag positions on the sphere as the end points of an arc generating a rotation. The user clicks down at $\hat{\mathbf{v}}_0$ and drags to $\hat{\mathbf{v}}_1$. As the

[1]Be careful not to confuse the unit quaternion hypersphere, where a single point represents a rotation, with this ordinary sphere, where a pair of points is required.

IBM ISBN 0-12-336155-9
Macintosh ISBN 0-12-336156-7

mouse is dragged, $\hat{\mathbf{v}}_1$ changes continuously, and so does the rotation. While dragging, we draw the changing arc and also the turning object.

Broken down into elementary steps, we do the following. Call the screen coordinates of the cursor at mouse down $\mathbf{s}_0 = (x_0, y_0, 0)$, the screen coordinates of the center of the Arcball $\mathbf{c}$, and its screen radius r. Compute $\mathbf{v}_0 = (\mathbf{s}_0 - \mathbf{c})/r$, and $z = \sqrt{1 - \|\mathbf{v}_0\|^2}$. Then $\hat{\mathbf{v}}_0 = \mathbf{v}_0 + (0, 0, z)$ is our first point on the unit sphere. Do the same thing with the current cursor coordinates to get $\hat{\mathbf{v}}_1$, and with these two points compute the unit quaternion $q_{\text{drag}} = [\hat{\mathbf{v}}_0 \times \hat{\mathbf{v}}_1, \hat{\mathbf{v}}_0 \cdot \hat{\mathbf{v}}_1]$. We use the Arcball to manipulate an object's orientation, which at mouse down we save as a quaternion, q_{down}. While dragging, we compute the object's current orientation as $q_{\text{now}} = q_{\text{drag}} q_{\text{down}}$. So long as the mouse button is held we use the same $\hat{\mathbf{v}}_0$ and q_{down}; upon release we permanently update the object's orientation to q_{now}.

◇ Arcball Properties ◇

Arcball's most important properties are hard to convey in print: it has a good "feel" and can be mastered in minutes. This is partly because the object motion mimics the mouse motion. If you drag across the center of the sphere, the object rotates in the direction the mouse moves. If you drag around the edge of the sphere, the object rotates around z in the same direction. But there is more to Arcball. With a single mouse stroke it is possible to rotate 360° around any axis. In fact, opposite points on the edge give the same rotation, so it is possible to wrap around and keep turning. And strokes add like vectors, which is truly remarkable since rotations do not commute.

This last property is the source of Arcball's path independence and requires a brief explanation. Consider two consecutive strokes, as in Figure 1.

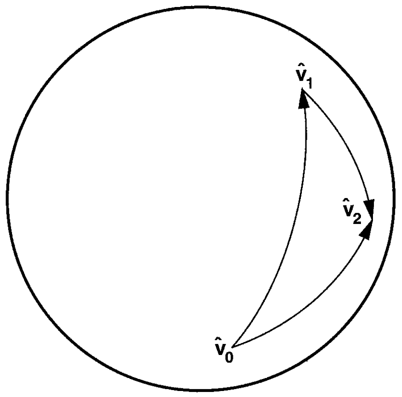
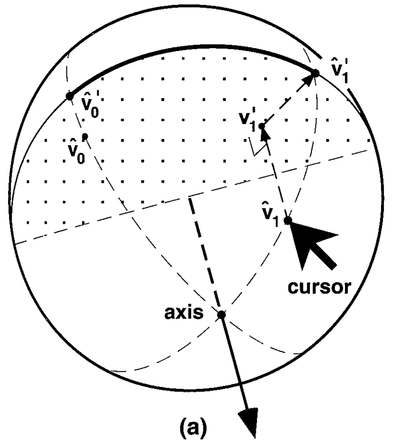


Figure 1. Arc addition.



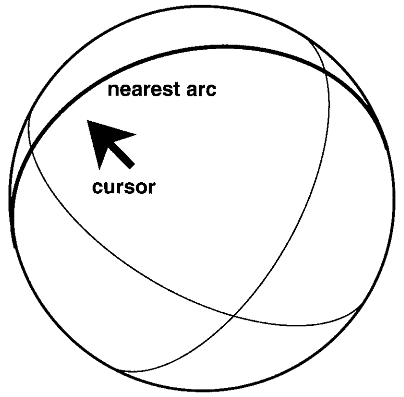


Figure 2. (a) Constraint implementation. (b) Constraint selection.

A stroke from $\hat{\mathbf{v}}_0$ to $\hat{\mathbf{v}}_1$ followed by a stroke from $\hat{\mathbf{v}}_1$ to $\hat{\mathbf{v}}_2$ gives the same effect as a direct stroke from $\hat{\mathbf{v}}_0$ to $\hat{\mathbf{v}}_2$. That's because the composite quaternion is $(\hat{\mathbf{v}}_2\hat{\mathbf{v}}_1^{-1})(\hat{\mathbf{v}}_1\hat{\mathbf{v}}_0^{-1}) = \hat{\mathbf{v}}_2\hat{\mathbf{v}}_0^{-1}$. The benefit is a more forgiving interface with a solid feel. Once you start dragging, where the mouse is positioned matters, but not how you got there. There is no permanent penalty for losing a mouse sample, which is often hard to avoid; the behavior is like lossless incremental accumulation. Path independence also makes displaying an arc meaningful, since it really does show you the cumulative effect of your drag.

Like Hanson, we can also use a pair of controllers to turn objects in 4D. The complexity of rotations grows with the square of the dimension, giving in 4D 6 degrees of freedom. We can use an arbitrary quaternion, p, to describe a point, and a pair of unit quaternions, u and v, to describe a rotation. In 3D, we use the formula upu^{-1}; a 4D version is $uvpu^{-1}$. Adjust u with one Arcball, and v with the other.

◇ Adding Constraints ◇

We can now rotate with full freedom, but sometimes we want less. Fortunately, Arcball can easily be augmented with axis constraints.[2] (See Figure 2a.) To implement this, take your original Arcball points, subtract their components parallel to your chosen axis, and renormalize onto the sphere. Call the unit axis vector $\hat{\mathbf{a}}$; then compute $\mathbf{v}_0' = \hat{\mathbf{v}}_0 - (\hat{\mathbf{v}}_0 \cdot \hat{\mathbf{a}})\hat{\mathbf{a}}$, and $\hat{\mathbf{v}}_0' = \mathbf{v}_0'/\|\mathbf{v}_0'\|$. Do the same for $\mathbf{v}_1$. If either $\mathbf{v}_0'$ or $\mathbf{v}_1'$ ends up with negative z, negate

[2]Where you get an axis is up to you. It could be a coordinate axis, a surface normal, a body principal axis of inertia, a light reflection direction, or whatever.

that vector to its opposite on the front hemisphere. Using these new points instead of the originals to compute q_{drag}, your rotation is now constrained.

Here's an easy way to pick one axis from a small set of choices. Signal constraint mode by holding down, say, the [SHIFT] key. Have the controller pop up arcs superimposed on the Arcball, one for each of your axes. As you move the mouse around before clicking, the closest arc should be highlighted. (See Figure 2b.) When you click down with the mouse, you are constrained to the axis for the closest arc, and the other arcs disappear. There is both a visual clue (seeing pop-up arcs) and a kinesthetic clue (holding down the [SHIFT] key) that you are in constraint mode. When you release only the mouse button, you stay in constraint mode and are again shown all the arc choices. When you release the [SHIFT] key, you return to free mode, signaled by having the constraint arcs disappear. If you have different axis sets (object axes, camera axes, et cetera), you can hold down different keys to signal constraint mode.

◇ Code ◇

```
/***** BallMath.h - Essential routines for Arcball.  *****/
#ifndef _H_BallMath
#define _H_BallMath
#include "BallAux.h"

HVect MouseOnSphere(HVect mouse, HVect ballCenter, double ballRadius);
HVect ConstrainToAxis(HVect loose, HVect axis);
int NearestConstraintAxis(HVect loose, HVect *axes, int nAxes);
Quat Qt_FromBallPoints(HVect from, HVect to);
void Qt_ToBallPoints(Quat q, HVect *arcFrom, HVect *arcTo);
#endif
/***** EOF *****/
/***** BallAux.h - Vector and quaternion routines for Arcball. *****/
#ifndef _H_BallAux
#define _H_BallAux

typedef int Bool;
typedef struct {float x, y, z, w;} Quat;
enum QuatPart {X, Y, Z, W, QuatLen};
typedef Quat HVect;
typedef float HMatrix[QuatLen][QuatLen];

extern Quat qOne;
HMatrix *Qt_ToMatrix(Quat q, HMatrix out);
Quat Qt_Conj(Quat q);
Quat Qt_Mul(Quat qL, Quat qR);
HVect V3_(float x, float y, float z);
float V3_Norm(HVect v);
HVect V3_Unit(HVect v);
HVect V3_Scale(HVect v, float s);
HVect V3_Negate(HVect v);
```