Com S 437
Flavia Cavalcanti

The ArcBall Technique

The purpose of this tutorial is to help you to better understand how to implement the ArcBall technique. It is expected that you have a basic understanding of linear algebra and that you are familiar with coding, as this tutorial will not focus on coding the technique itself, but rather how to retrieve a rotation matrix by using the arc ball technique.

The ArcBall technique is a very useful algorithm to rotate 3D objects, as it provides an intuitive method to do so. It receives a mouse movement as an input and uses the mouse coordinates to output a rotation matrix. You might be wondering why one would bother with such a technique, when assigning different rotations to different keyboard keys appears to be a more intuitive approach. There are actually many reasons to choose arcball over the keyboard, and we will discuss them soon, but first consider the following scenario.

You are playing a game where you have to pilot a ship through different waypoints; every ship movement is linked to a key, including its rotations and its ascent and descent in space. This ideology would result in 6 different movement keys (forward thrust, reverse thrust, turn port (left), turn starboard (right), ascend, and descend), not including any other action keys that had yet to be assigned. This, interestingly enough, isn't a hypothetical situation but is in fact the control system used in the game, Guns of Icarus.

Figure 1. The convoluted keyboard controls of Guns of Icarus. As you can see, the ship movement is hooked up to 6 different keys.

Assuming you were not making a game with confusing controls just for the sake of wanting to increase the level of difficulty of your game, you might want to consider using arcball to get rid of at least four of those keys. However, before we can begin to understand the intricacies of such a technique, we must first understand a few key concepts.

Rotations Around a Vector ê

Imagine that we have a point P, which we want to rotate around an axis ê by θ degrees, as depicted in figure 2. The first step [2] is to describe the rotated point p' as a function of p, using an appropriate base. For this purpose, consider the following orthogonal base: $\hat{e}, p_{\perp}, v, \left\|\hat{e}\right\| = 1$

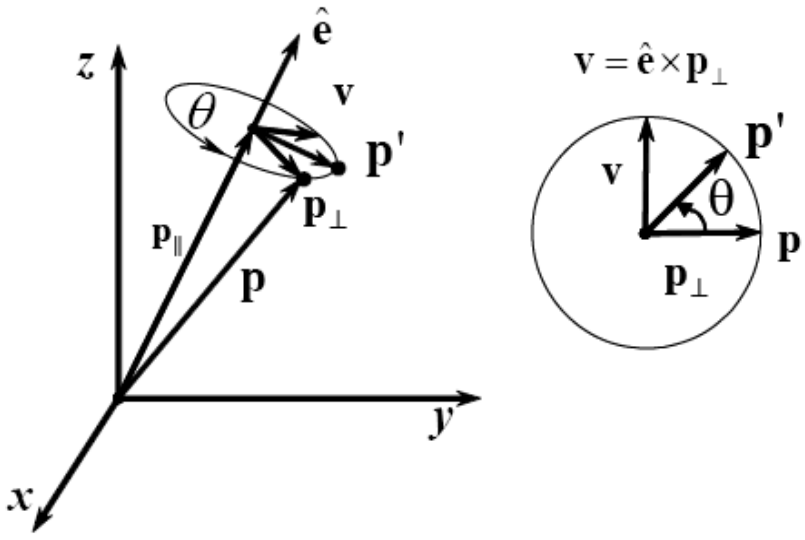


Figure 2. The vectors involved in rotation.

From the information available to us, we can write p and p' as:

$$p = p_{\perp} + p_{\parallel}$$

$$p' = R(p_{\perp}) + R(p_{\parallel})$$

$$v = \hat{e} \times p_{\perp}$$

$$p' = p_{\parallel} + (\cos\theta)p_{\perp} + (\sin\theta)v$$

$$p_{\parallel} = (\hat{e} \cdot p)\hat{e}$$

$$p_{\perp} = p - (\hat{e} \cdot p)\hat{e}$$

$$v = \hat{e} \times p_{\perp} = \hat{e} \times (p - (\hat{e} \cdot p)\hat{e}) = \hat{e} \times p - (\hat{e} \cdot p)\hat{e} \times \hat{e} = \hat{e} \times p$$

$$p' = (\hat{e} \cdot p)\hat{e} + (\cos\theta)(p - (\hat{e} \cdot p)\hat{e}) + (\sin\theta)(\hat{e} \times p)$$

$$p' = (\cos\theta)p + (1 - \cos\theta)(\hat{e} \cdot p)\hat{e} + (\sin\theta)(\hat{e} \times p) \quad \textbf{[equation 1]}$$

As we can see, this is the linear transformation that rotates a given point p. We have yet to construct a rotation matrix in the x, y, z coordinate system. To do so, we just need to apply this transformation to each vector of the canonical base.

The columns of M are given by: p'(1, 0, 0), p'(0,1, 0), p'(0, 0, 1):

$$\mathbf{M} = \begin{bmatrix} \cos\theta + (1-\cos\theta)e_x^2 & e_y e_x(1-\cos\theta) - e_z\sin\theta & e_z e_x(1-\cos\theta) + e_y\sin\theta & 0 \\ e_x e_y(1-\cos\theta) + e_z\sin\theta & \cos\theta + (1-\cos\theta)e_y^2 & e_z e_y(1-\cos\theta) - e_x\sin\theta & 0 \\ e_x e_z(1-\cos\theta) - e_y\sin\theta & e_y e_z(1-\cos\theta) - e_x\sin\theta & \cos\theta + (1-\cos\theta)e_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Gimbal Lock

Usually when people think of matrix rotations, they immediately jump to the conclusion that they have to use Euler angles. The problem is that there are quite a few

problems associated with using Euler Angles in rotations, and among these is the so-called Gimbal Lock[3, 4].

Gimbal Lock is a term that is quite often associated with aircraft systems that use gyroscopes. Gimbals are used in gyroscopes to determine the orientation of a given aircraft. Gimbal lock occurs when there is a loss of one degree of freedom in a 3-gimbal mechanism. Consider the figure below.

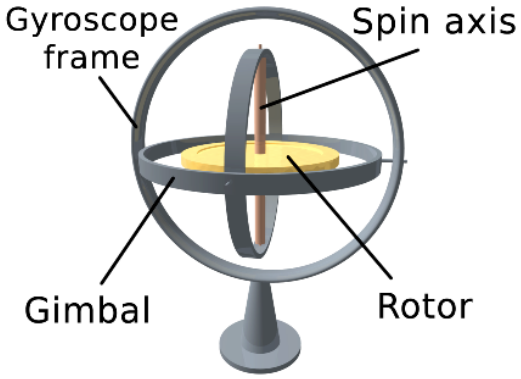


Figure 3. A gyroscope with 3 gimbals.

This figure depicts a representation of a gyroscope, where the three loops around the rotor are its gimbals. If two or more axes of the gimbals were driven into a parallel configuration, then the system would lock into place.

Gimbal lock can be quite a serious situation, especially in aircrafts. Aircrafts usually use an Inertial Mechanism Unit for navigation, where given a starting point, such a system can record where you are in relation to that point at any time. Each gyroscope, however, has a "null point", where the gyro lines up perfectly with its gimbals. If all three enter this state at once, then the platform instantly loses its knowledge of where it is. Entering a "gimbal lock" state.

The term Gimbal Lock is used in computer graphics to reference a problem that only occurs with Euler angles. Imagine that we want to perform a rotation along the x, y, and z-axis, dubbed as R_X, R_Y, and R_Z. A general rotation R is calculated by multiplying these three matrices to retrieve the resulting rotation matrix.

$$R(\theta_1,\theta_2,\theta_3) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_1 & \sin\theta_1 & 0 \\ 0 & -\sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta_2 & 0 & -\sin\theta_2 & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta_2 & 0 & \cos\theta_2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta_3 & \sin\theta_3 & 0 & 0 \\ -\sin\theta_3 & \cos\theta_3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

With the resulting matrix multiplication being equal to:

$$R(\theta_1,\theta_2,\theta_3) = \begin{pmatrix} c_2c_3 & c_2s_3 & -s_2 & 0 \\ s_1s_2c_3 - c_1s_3 & s_1s_2s_3 + c_1c_3 & s_1c_2 & 0 \\ c_1s_2c_3 + s_1s_3 & c_1s_2s_3 - s_1c_3 & c_1c_2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, s_i = \sin\theta_i, c_i = \cos\theta_i$$

Now consider a rotation around the y-axis by 90 degrees.

$$R(\theta_1,\frac{\pi}{2},\theta_3) = \begin{pmatrix} 0 & 0 & -1 & 0 \\ \sin(\theta_1-\theta_3) & \cos(\theta_1-\theta_3) & 0 & 0 \\ \cos(\theta_1-\theta_3) & -\sin(\theta_1-\theta_3) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, s_2 = 1, c_2 = 0$$

As can be noted, the term θ_2 has disappeared in the final matrix, leading to a loss of freedom. The object is now locked, that is, cannot rotate around the y-axis.

Gimbal lock occurs because the Euler angles are being changed, in a fixed order, individually, allowing the axes to line up. This situation is rectified in aircraft navigation systems by adding a “fourth gimbal”, which is not allowed to align. Therefore, the solution to this problem would be the addition of such a “fourth gimbal” to rotations, which leads us to quaternions.

Quaternions

Quaternions[2] are complex numbers on R^4, where a quaternion q is 4-tuple of numbers:

q = a + bi + cj + dk = (s, v),where s is a real component and v is the vector that represents the imaginary component. It is possible to define multiplication and addition of quaternions:

$$q_1 = (s_1, v_1),\ q_2 = (s_2, v_2)$$

$$q_1 q_2 = \left(s_1 s_2 - v_1 \cdot v_2,\ s_1 v_2 + s_2 v_1 + v_1 \times v_2\right)$$

$$q_1 + q_2 = (s_1 + s_2,\ v_1 + v_2)$$

The multiplication of two quaternions is a quaternion and quaternion multiplication is not commutative, therefore, $q_1.q_2 \neq q_2.q_1$. This property makes quaternions incredibly useful for object rotations.

The conjugate and reciprocal of a quaternion are obtained by inverting the imaginary axis:

$$\overline{q} = (s, -v) \quad \text{and} \quad q^{-1} = \frac{\overline{q}}{\|q\|^2}$$

$$q\overline{q} = (s, v)(s, -v) = s^2 + \|v\|^2 = s^2 + x^2 + y^2 + z^2 = \|q\|^2,$$

and its norm can be defined as:

$$\|q\| = \sqrt{q\overline{q}} = \sqrt{\overline{q}q} = \sqrt{s^2 + \|v\|^2}$$

To apply a rotation on a point p, take a pure quaternion p = (0, r) and a unit quaternion q = (s, v), where $q\overline{q} = 1$, and define: $R(p) = qpq^{-1}$

A unit quaternion can be written as:

$$q = \left(\cos(\theta),\ \sin(\theta)\hat{\mathbf{e}}\right),\ q^{-1} = (\cos(\theta), -\sin(\theta)\hat{\mathbf{e}}),\ \|\hat{\mathbf{e}}\| = 1,\ p = (0, r)$$

$$R_q(p) = qpq^{-1} = (0,\ (\cos^2\theta - \sin^2\theta)r + 2\sin^2\theta(\hat{e}\cdot r)\hat{e} + 2\cos\theta\sin\theta(\hat{e}\times r)) =$$

$$= (0,\ (\cos 2\theta)r + (1 - \cos 2\theta)(\hat{e}\cdot r)\hat{e} + \sin 2\theta(\hat{e}\times r)),$$

which is exactly the same as [equation 1], aside from a factor of 2 and p being written as r. Therefore, orientations can be parameterized as:

$$q = \left(\cos\left(\frac{\theta}{2}\right), \sin\left(\frac{\theta}{2}\right)\hat{\mathbf{e}} \right), \|\hat{\mathbf{e}}\| = 1$$

Quaternions can be used to apply rotations about an axis.

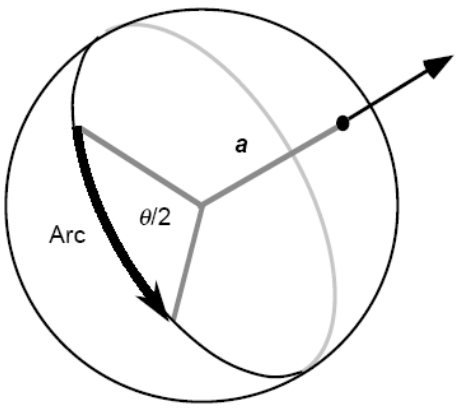
The rotation R, of a point p about an axis v, is given by $R(p) = qpq^{-1}$, where p = (0, r) is the point represented as a pure quaternion, and q = (s, v) is a unit quaternion representing the rotation angle and axis.

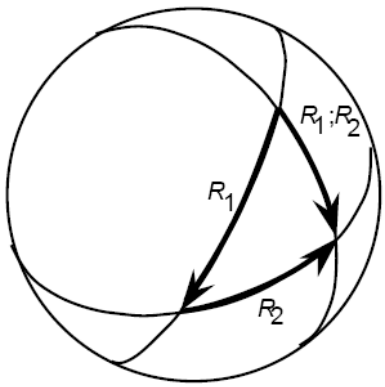
Using ArcBall Without Quaternions

As mentioned before, the reason why Arcball [1] is such an appealing technique is due to its intuitive user interface for the rotations of 3D objects. Even though this tutorial did emphasize a decent amount of quaternions, Arcball itself does not necessarily require them to function. You might be wondering why we'd bother to even mention them; however, you must understand that quaternions are an important aspect of 3D object rotations and most Arcball implementation do use quaternions.

The main reason is to avoid Gimbal lock in 3D animations, when interpolating orientations (e.g., by using key frames).

The following will describe an Arcball implementation that can be used with or without quaternions, as it is probably more intuitive to first timers.





Arc Interpretation **Arc Combination**

a – rotation axis

θ – rotation angle

Figure 4. Rotations with quaternions. Differently from rotations using Euler angles, there's only one way to interpolate from point A to point B, when using quaternions. Quaternions are also quite useful for rotations, as they can be easily stacked without leading to gimbal lock.

An intuitive implementation can be performed [7] by "inscribing" a hemisphere into the computer's window space, where the origin (0, 0) is in the upper left corner of the window, the x-axis runs along the screen width and the y-axis runs along the screen height. Take the lower right corner to be (w, h), or the width and height of our screen.

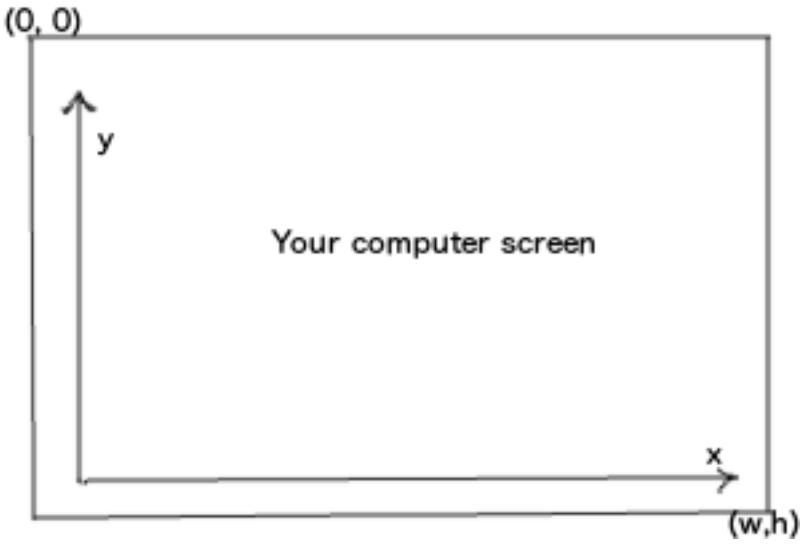


Figure 5. The representation of your computer screen as per this problem's specifications.

The trick is converting this 2D environment to 3D. To do so, it is possible to add a z component to the coordinate system, by considering a circle onto the screen area, and using the formula of a sphere.

$$x^2 + y^2 + z^2 = R^2 \Rightarrow z = \sqrt{R^2 - x^2 - y^2}$$

The function z (x, y)

considers the sphere centered at the middle of the screen with the maximum possible radius:

$$z(x,y) = \begin{cases} \sqrt{\left(\frac{h}{2}\right)^2 - \left(x - \frac{w}{2}\right)^2 - \left(y - \frac{h}{2}\right)^2} & \text{if } \left(x - \frac{w}{2}\right)^2 + \left(y - \frac{h}{2}\right)^2 < \left(\frac{h}{2}\right)^2 \\ 0 & \text{if } \left(x - \frac{w}{2}\right)^2 + \left(y - \frac{h}{2}\right)^2 \geq \left(\frac{h}{2}\right)^2 \end{cases}$$

Any point within the circle is mapped onto a sphere, and any point outside is mapped to the z = 0 plane.

This allows rotations to occur only when the selection is within the circle. Imagine that the user clicked on an area within the circle, the previously mentioned z-mapping function allows the definition of a start vector $\mathbf{p}_1 = (x, y, z) \in \mathbb{R}^3$ from which we can start constructing our rotation from. After the first click, the user can then begin dragging the mouse across the screen, translating p1 to p2.

Supposing the user reaches a new position given by (x', y') coordinates, we'd have to remap this coordinate using the aforementioned function, obtaining the following vector: $\mathbf{p}_2 = (x', y', z(x', y'))$

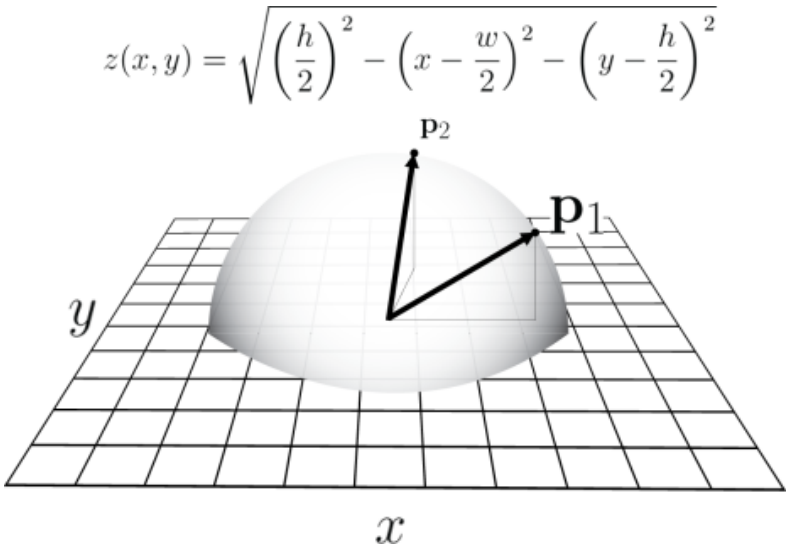


Figure 6. The so-called hemisphere on our screen space, where P1 is the initial vector and P2 is the final vector.

We now have access to a start and final vector. However, we still have yet to define a rotation axis and a rotation angle.

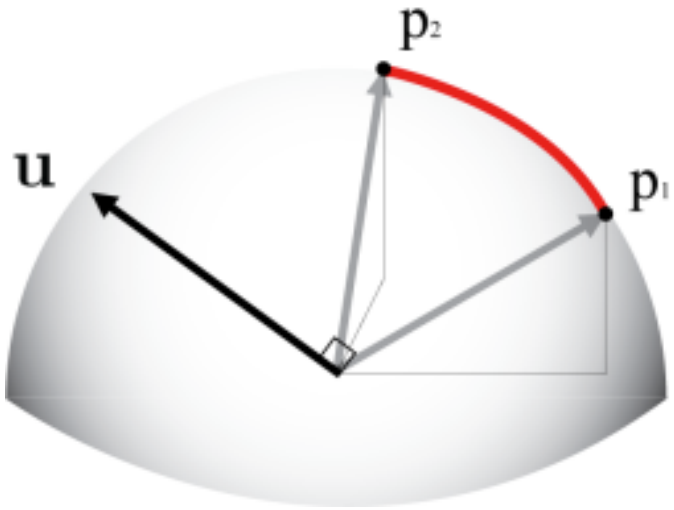


Figure 7. The image above depicts the required vectors to successfully perform the rotation. We need a start, final, and a rotation axis vector.

The next step is to compute the rotation in terms of an $\mathbb{R}^{3\times3}$ orthogonal rotation matrix by calculating the angle between P1 and P2. The simplest way to do so is to use the dot product of P1 and P2 to find angle θ:

$$\theta = \arccos\left(\frac{\mathbf{p}_1 \cdot \mathbf{p}_2}{||\mathbf{p}_2|| \cdot ||\mathbf{p}_1||}\right)$$

And the normalized cross product of P1 and P2 to find the axis of rotation:

$$\mathbf{u} = \frac{\mathbf{p}_1 \times \mathbf{p}_2}{||\mathbf{p}_2|| \cdot ||\mathbf{p}_1||}$$

We now have all of the information we need to perform a rotation. We currently have two options; use the rotation matrix that was defined in the beginning of this tutorial or using a quaternion to perform the rotation.

Option 1 using a rotation matrix. Substitute theta for the rotation angle and e for the corresponding x, y, or z component of the rotation axis.

$$\mathbf{M} = \begin{bmatrix} \cos\theta + (1-\cos\theta)e_x^2 & e_y e_x(1-\cos\theta) - e_z\sin\theta & e_z e_x(1-\cos\theta) + e_y\sin\theta & 0 \\ e_x e_y(1-\cos\theta) + e_z\sin\theta & \cos\theta + (1-\cos\theta)e_y^2 & e_z e_y(1-\cos\theta) - e_x\sin\theta & 0 \\ e_x e_z(1-\cos\theta) - e_y\sin\theta & e_y e_z(1-\cos\theta) - e_x\sin\theta & \cos\theta + (1-\cos\theta)e_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Option 2 using quaternions. Substitute θ for the rotation angle and *e* with the rotation axis that we just found.

$$q = \left(\cos\left(\frac{\theta}{2}\right), \sin\left(\frac{\theta}{2}\right)\hat{\mathbf{e}}\right)$$

I tend to be more partial to using quaternions, as you do not have to keep track of 16 different numbers. After choosing your desired method, you can then apply the rotation matrix to your model and enjoy the results of being able to rotate your 3D object solely with your mouse.

References

[1] *ARCBALL: A User Interface for Specifying Three-Dimensional Orientation Using a Mouse* - Ken Shoemake

[2] *Advanced Animation and Rendering Techniques* – Alan Watt

[3] https://www.youtube.com/watch?v=zc8b2Jo7mno

[4]https://sundaram.wordpress.com/2013/03/08/mathematical-reason-behind-gimbal-lock- in-euler-angles/

[5] http://www.universetoday.com/119984/13-more-things-that-saved-apollo-13-part-9-avoiding-gimbal-lock/

[6] https://www.youtube.com/watch v=mHVwd8gYLnI&nohtml5=False

[7]https://braintrekking.wordpress.com/2012/08/21/tutorial-of-arcball-without-quaternions/

[8]https://sureshemre.wordpress.com/2014/07/19/a-rotation-equals-two- reflections/

[9]http://math.hws.edu/eck/cs424/notes2013/webgl/skybox-and-reflection/simple-rotator.js

[10] http://en.wikipedia.org/wiki/Euler_angles