Article

Working with Quaternions

Rotate points around the surface of a sphere, and interpolate between them.

Overview

Quaternions are defined by a scalar (*real*) part, and three imaginary parts collectively called the *vector* part. Quaternions are often used in graphics programming as a compact representation of the rotation of an object in three dimensions.

The length of a quaternion is the square root of the sum of the squares of its components. For example, consider a quaternion specified by the following code:

```
let ix = 1.0
let iy = 4.0
let iz = 8.0
let r = 9.0

let q = simd_quatd(ix: ix, iy: iy, iz: iz, r: r)
```

The length of the quaternion can be calculated manually with `sqrt(ix*ix + iy*iy + iz*iz + r*r)`, or more simply accessed through its `length` property. Quaternions with a length of one are called *unit quaternions* and can represent rotations in 3D space. You can easily convert a nonunit quaternion representing a rotation into a unit quaternion by normalizing its axes. The following code shows q1, which contains rotations around all three axes with a length greater than 1, and q2, which contains the same rotation but has a length of 1 and is, therefore, suitable for applying a rotation to a 3D coordinate:

```
let axis = simd_double3(x: -2,
                        y: 1,
                        z: 0.5)
```

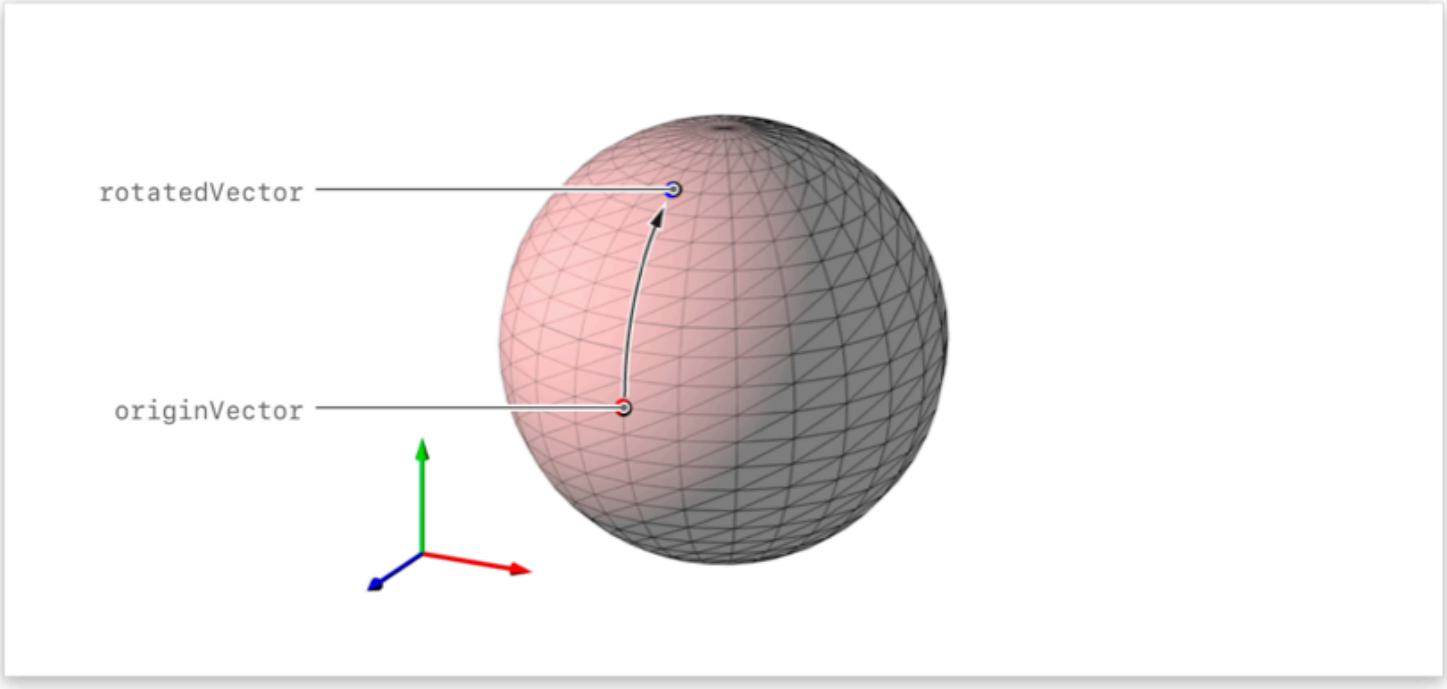
```
// `q1` length = 2.29128
let q1 = simd_quatd(angle: .pi,
                    axis: axis)

// `q2` length = 1
let q2 = simd_quatd(angle: .pi,
                    axis: simd_normalize(axis))
```

Quaternions have some advantages over matrices. For example, they're smaller: A 3 x 3 matrix of floats is 48 bytes, and a single-precision quaternion is 16 bytes. They also can offer better performance: Although a single rotation using a quaternion is a little slower than one using a matrix, when combining actions, quaternions can be up to 30% faster.

The following examples show a few common uses of quaternions.

Rotate a Point Around a Sphere

The following illustration shows a point, defined by `originVector`, rotated over the surface of a sphere by 60° about the x axis.



To apply this rotation, you define the vector to be rotated and the quaternion that represents the rotation:

```
func degreesToRadians(_ degrees: Float) -> Float {
    return degrees * .pi / 180
```

```
}

let originVector = simd_float3(x: 0, y: 0, z: 1)

let quaternion = simd_quatf(angle: degreesToRadians(-60),
                            axis: simd_float3(x: 1,
                                              y: 0,
                                              z: 0))
```

The rotation of the vector by a quaternion is known as an *action*; to apply the rotation to `originVector`, you call the `act(_:)` method:

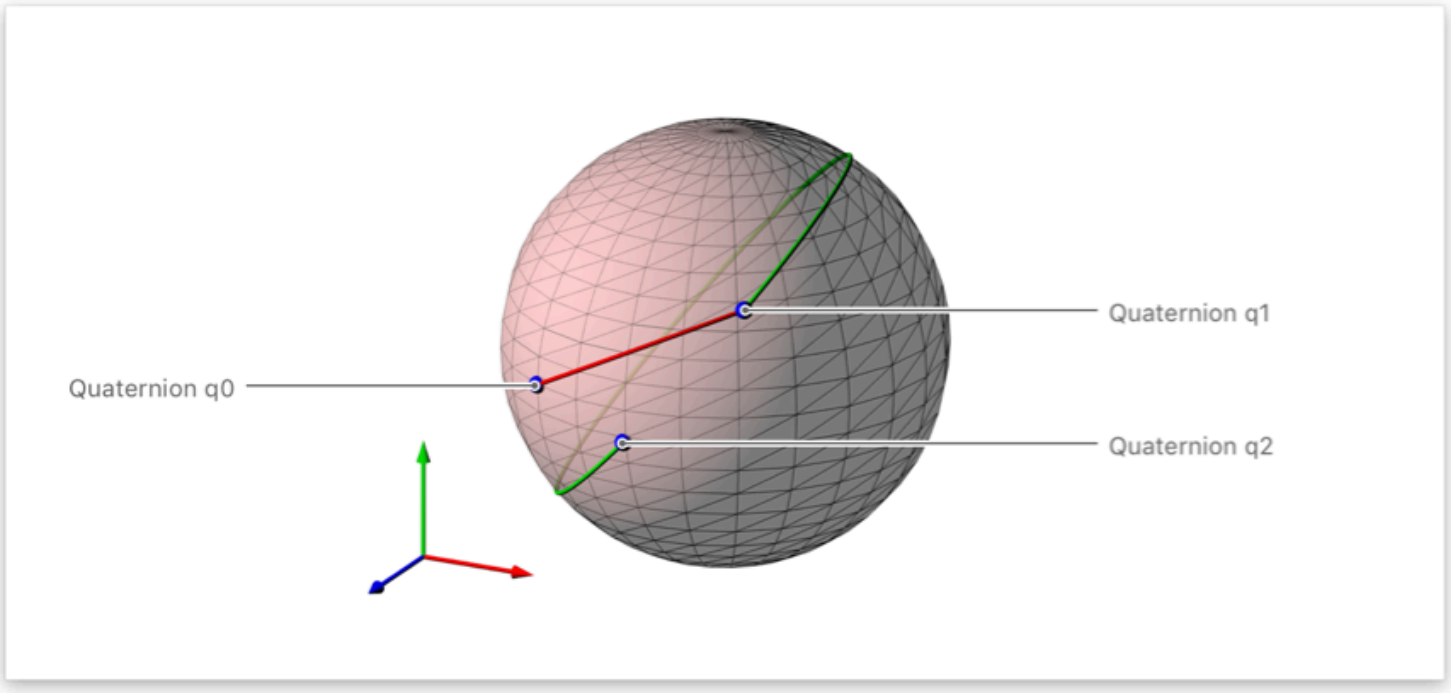
```
let rotatedVector = quaternion.act(originVector)
```

`rotatedVector` is now at the position of the blue point in the illustration above.

Interpolate Between Two Points on a Sphere

One of the advantages quaternions have over matrices when representing rotations is that they simplify interpolation between rotated coordinates.

The following image shows the spherical linear interpolation along the shortest arc between `q0` and `q1`, and along the longest arc between `q1` and `q2`.



The following code shows how the three points are defined in the preceding image:

```
let origin = simd_float3(0, 0, 1)

let q0 = simd_quatf(angle: .pi / 6,
                    axis: simd_normalize(simd_float3(x: 0,
                                                     y: -1,
                                                     z: 0)))
let u0 = simd_act(q0, origin)

let q1 = simd_quatf(angle: .pi / 6,
                    axis: simd_normalize(simd_float3(x: -1,
                                                     y: 1,
                                                     z: 0)))
let u1 = simd_act(q1, origin)

let q2 = simd_quatf(angle: .pi / 20,
                    axis: simd_normalize(simd_float3(x: 1,
                                                     y: 0,
                                                     z: -1)))
```

The `simd_slerp(_:_:_:)` function linearly interpolates along the shortest arc between two quaternions. The following code calls `simd_slerp(_:_:_:)` with small increments to its `t` parameter, adding a line segment at each interpolated value to build the short arc between `q0` and `q1` shown in the preceding image:

```
for t: Float in stride(from: 0, to: 1, by: 0.001) {
    let q = simd_slerp(q0, q1, t)
    // code to add line segment at `q.act(origin)`
}
```

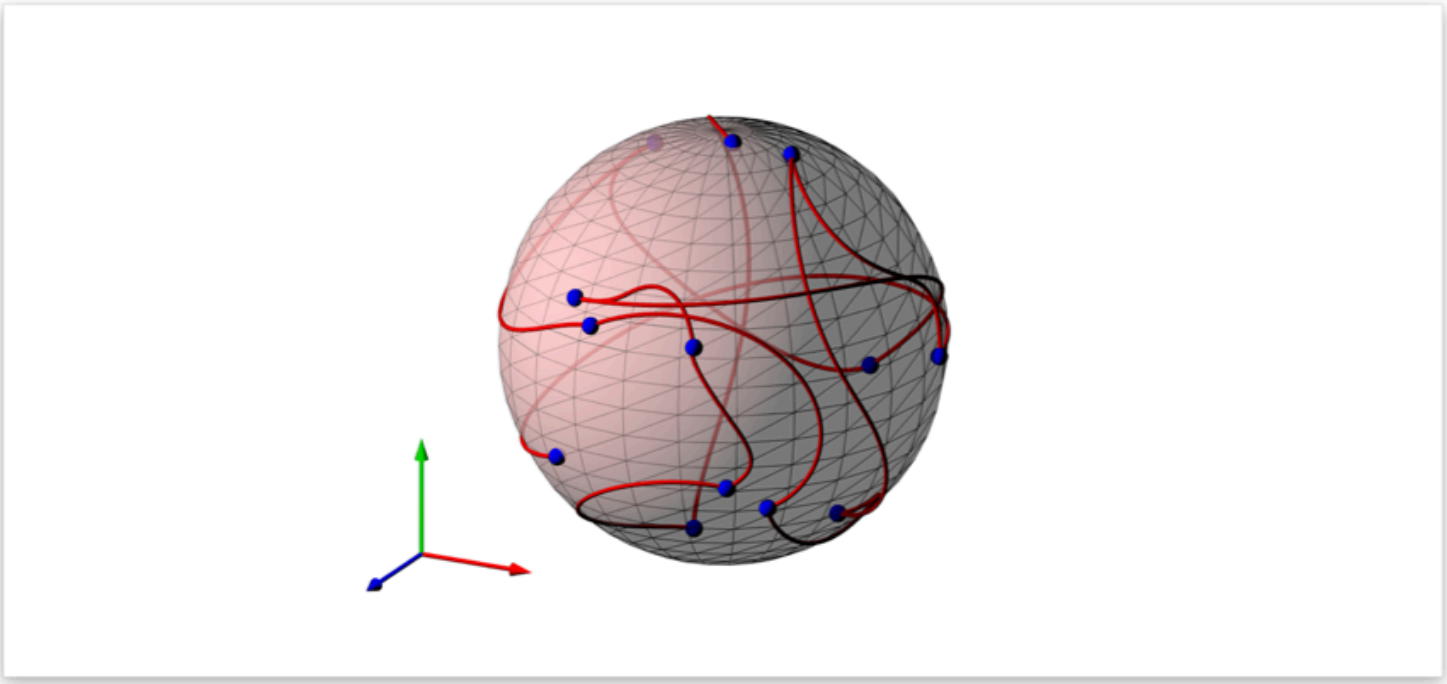
The `simd_slerp_longest(_:_:_:)` function linearly interpolates along the longest arc between two quaternions. The following code calls `simd_slerp_longest(_:_:_:)` with small increments to its `t` parameter, adding a line segment at each interpolated value to build the long arc between `q1` and `q2` shown in the preceding image:

```
for t: Float in stride(from: 0, to: 1, by: 0.001) {
    let q = simd_slerp_longest(q1, q2, t)
    // code to add line segment at `q.act(origin)`
}
```

Interpolate Between Multiple Points on a Sphere

To interpolate between multiple quaternions that define positions on the surface of a sphere, the simd library provides the `simd_spline(_:_:_:_:)` function.

The following image illustrates a series of quaternions that define positions on the surface of a sphere, shown as points, and a line that shows the smooth interpolation between them.



Much like `simd_slerp(_:_:_:)`, `simd_spline(_:_:_:_:)` accepts the two quaternions to interpolate between, but also requires the surrounding two quaternions. Given an array of quaternions named `rotations`, the following code iterates over each element, adding a line segment at each interpolated value to build the smooth line shown in the preceding image:

```
let rotations: [simd_quatf] = ...

for i in 1 ... rotations.count - 3 {
    for t: Float in stride(from: 0, to: 1, by: 0.001) {
        let q = simd_spline(rotations[i - 1],
                            rotations[i],
                            rotations[i + 1],
                            rotations[i + 2],
                            t)
        // code to add line segment at `q.act(origin)`
    }
}
```

See Also

Vectors, Matrices, and Quaternions

Working with Vectors

Use vectors to calculate geometric values, calculate dot products and cross products, and interpolate between values.

Working with Matrices

Solve simultaneous equations and transform points in space.

Rotating a cube by transforming its vertices

Rotate a cube through a series of keyframes using quaternion interpolation to transition between them.

simd

Perform computations on small vectors and matrices.

vForce

Perform transcendental and trigonometric functions on vectors of any length.