Advanced Animation Programming

GPR-450
Daniel S. Buckstein

Quaternions for Animation
Week 4

License

Quaternions

- Review of rotation matrices and their issues
- Interpolation over an arc
  - NLERP & SLERP
- Quaternions and applications
  - Operations
  - Quaternion SLERP
  - Comparison with matrices
  - Applications

Rotation Matrices

- 3x3 matrices can be used to represent *rotations in 3D*

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Rotation Matrices

- ***Rodrigues' rotation formula*** (axis-angle):

$$R = I + (\sin\theta)S + (1 - \cos\theta)S^2$$

where I is the 3x3 identity matrix,

θ is the angle of rotation, and

$S = \begin{bmatrix} 0 & -\hat{n}_z & \hat{n}_y \\ \hat{n}_z & 0 & -\hat{n}_x \\ -\hat{n}_y & \hat{n}_x & 0 \end{bmatrix}$, where $\hat{n}$ is the normalized axis of rotation

Rotation Matrices

- **Concatenation**:
- Also known as matrix multiplication
- ***Non-commutative***: written order matters!

$$AB \neq BA$$

- ***Associative***: chaining more than 2 matrices, does not matter which concatenation happens first

$$(AB)C = A(BC)$$

Rotation Matrices

- **Concatenation**:
- Easy way to multiply rotation matrices:
- For the matrix product $C = AB$
- For each element $C_{r,c}$ where r is the row and c is the column…
- …take the *dot product* of row r in matrix A (the left) and column c in matrix B (right)

Rotation Matrices

- **Concatenation**:
- When describing rotations, the first rotation is written on the *right*
- E.g. $R = R_0 R_1$
- In this example, the rotation R_1 will occur first
- This is not the same as $R_1 R_0$
- Easily demonstrated with real objects!

Rotation Matrices

- **Inverse**: finding the inverse of a 3x3 matrix is a time-consuming process
- PRO TIP: For *rotation matrices*, the transpose is also the inverse!!! $R^{-1} = R^T$
- How do you know if a 3x3 matrix is a rotation matrix???

Rotation Matrices

- A 3x3 matrix can be used as a rotation if its *determinant* is *equal to 1*
- **Determinant**: For a 3x3 matrix *A*

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

- The determinant is

$$\det(A) = a(ei - fh) + b(fg - di) + c(dh - eg)$$

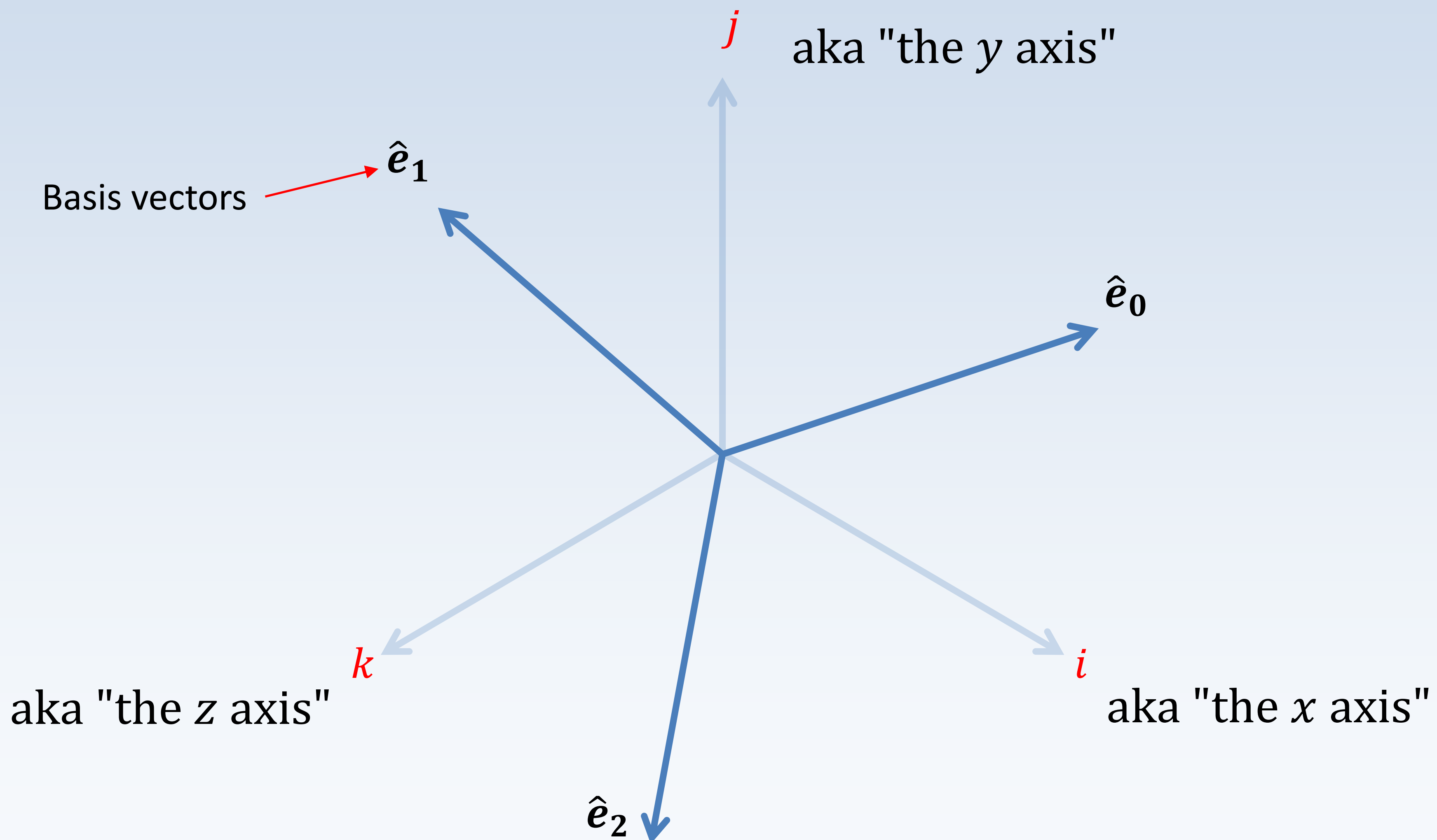
Rotation Matrices

- A 3x3 rotation matrix can be written as three normalized column vectors instead of nine elements:

$$R = [\hat{e}_0 \quad \hat{e}_1 \quad \hat{e}_2]$$

- These are the ***basis vectors*** for a coordinate system!

Rotation Matrices

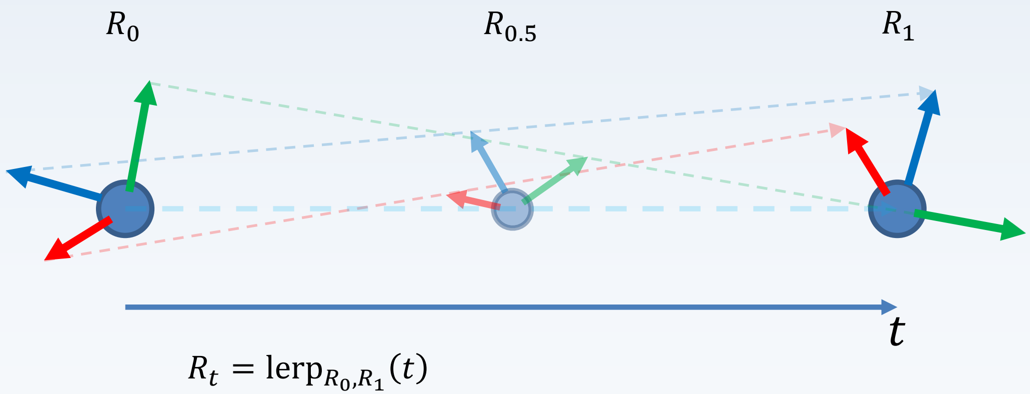


Rotation Matrices

- This is generally a tough concept to grasp
- One helpful way to understand basis vectors is to remove the notion of *absolute direction*
- Everything in animation/physics is *relative*
- Think of basis vectors as the *directions* ***relative*** to their parent coordinate frame!

Rotation Matrices

- Linearly interpolating rotation matrices:
- This has the same effect as applying *scale* simultaneously…
- Here's a graphical example (over time):



Rotation Matrices

- Since a rotation matrix can be thought of as 3 *unit* vectors…
- …LERP on a matrix is the same as LERP on 3 vectors simultaneously…
- Therefore the result is 3 *shorter* vectors, indicating *scale*

Rotation Matrices

- Rotation matrices are usually constructed from Euler angles
- Three separate rotations, one for each axis, multiplied together to give us one rotation
- *HUGE* problem with this…???
- ***Gimbal lock***

https://en.wikipedia.org/wiki/Gimbal_lock

Rotation Matrices

- Matrices are *required* for rendering…
- …but they are terrible for animation/physics…
- Cannot LERP matrices without consequences

- If only there was a ***tool*** to alleviate gimbal lock and *animate rotations* without the headache…

Recap: LERP

- The fundamental formula for everything!
- ***<u>Linear interpolation (LERP)</u>***

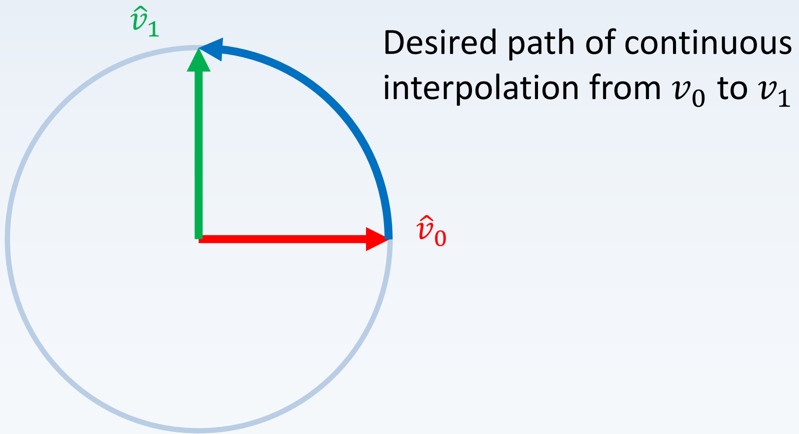
$$\mathrm{lerp}_{v_0,v_1}(t) = v_0 + t(v_1 - v_0)$$

where

t is the interpolation parameter
v_0 is the result at $t = 0$ (treat as constant)
v_1 is the result at $t = 1$ (ditto)

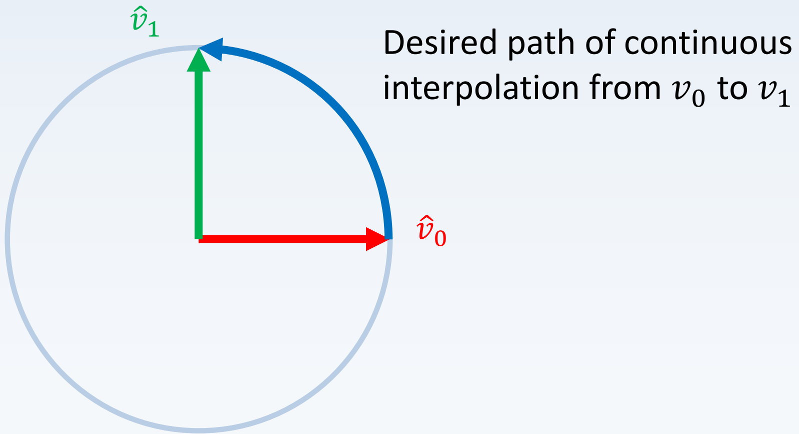
Arc Interpolation

- Given two *direction vectors* (2D or 3D for now) that we want to interpolate over an arc
- The desired arc interpolation path from *v0* to *v1* (controlled by *t* parameter) looks like this:



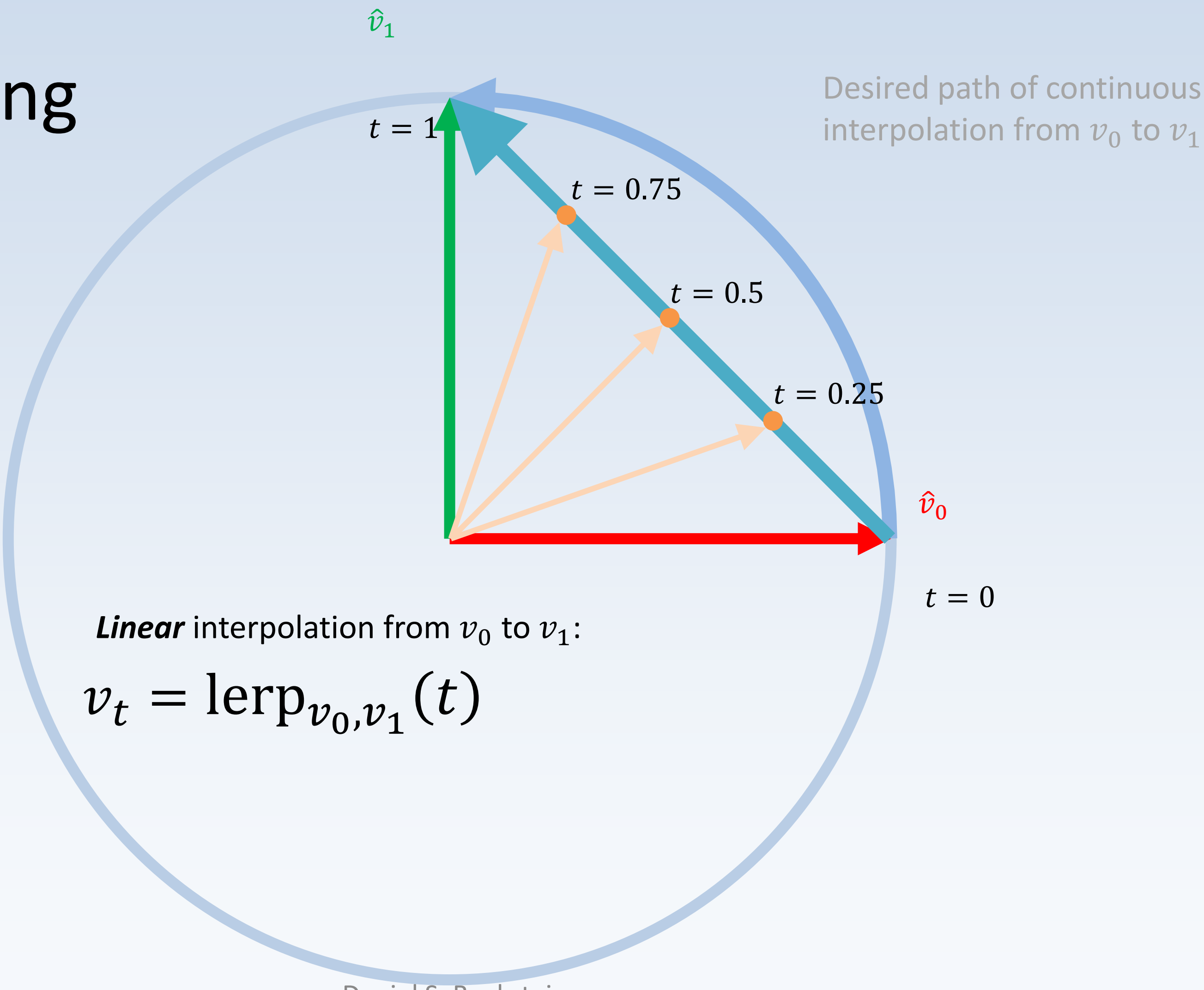
Arc Interpolation

- How do we <u>*simulate*</u> this interpolation???
  - (using the same interpolation rules we are already familiar with!)
- We know how to use LERP
- When *t=0* result is *v0*, when *t=1* result is *v1*



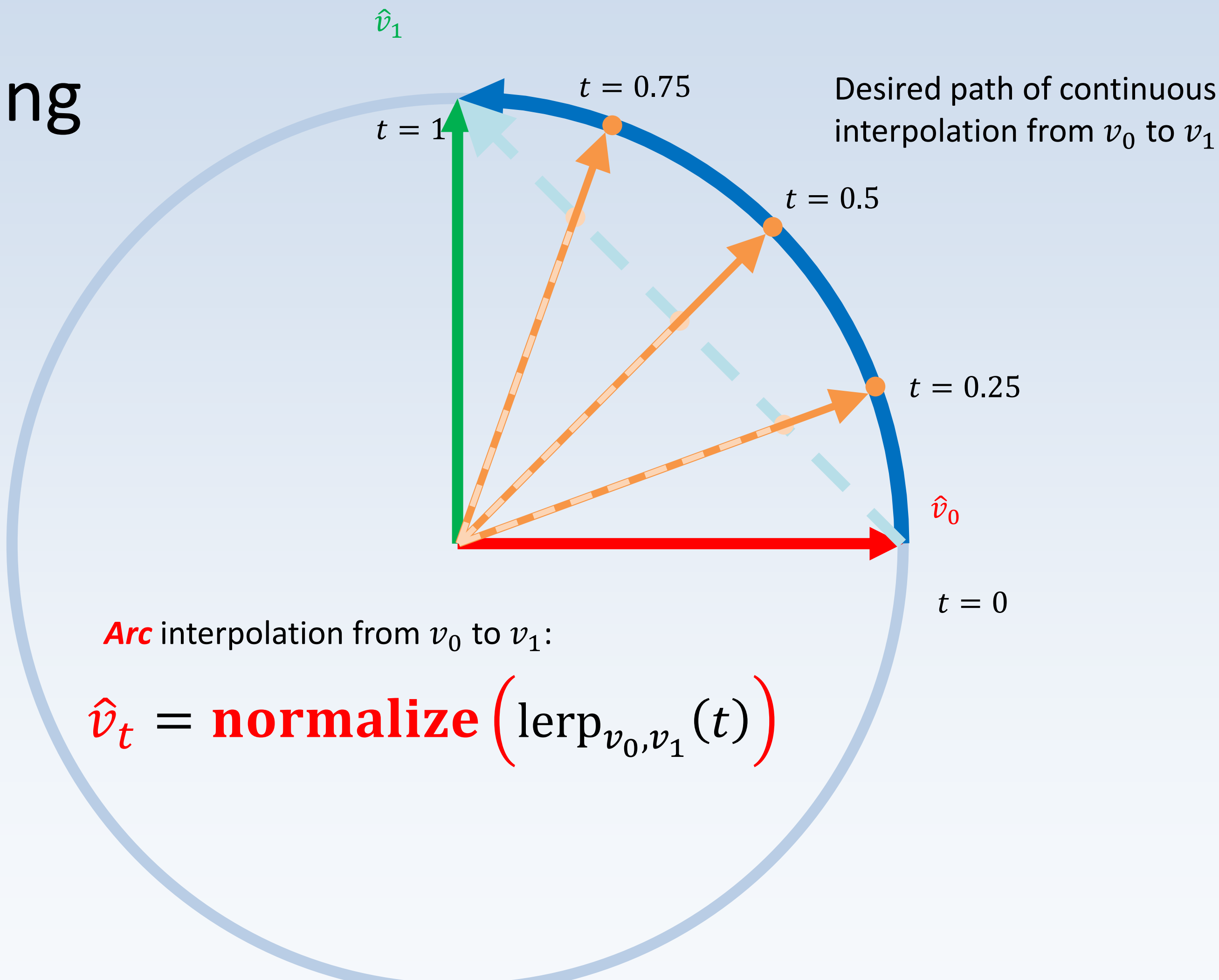
Arc Interpolation

- Simulating the arc:



Arc Interpolation

- Simulating the arc:



Arc interpolation from v_0 to v_1:

$$\hat{v}_t = \mathbf{normalize}\left(\mathrm{lerp}_{v_0,v_1}(t)\right)$$

NLERP: Normalized LERP

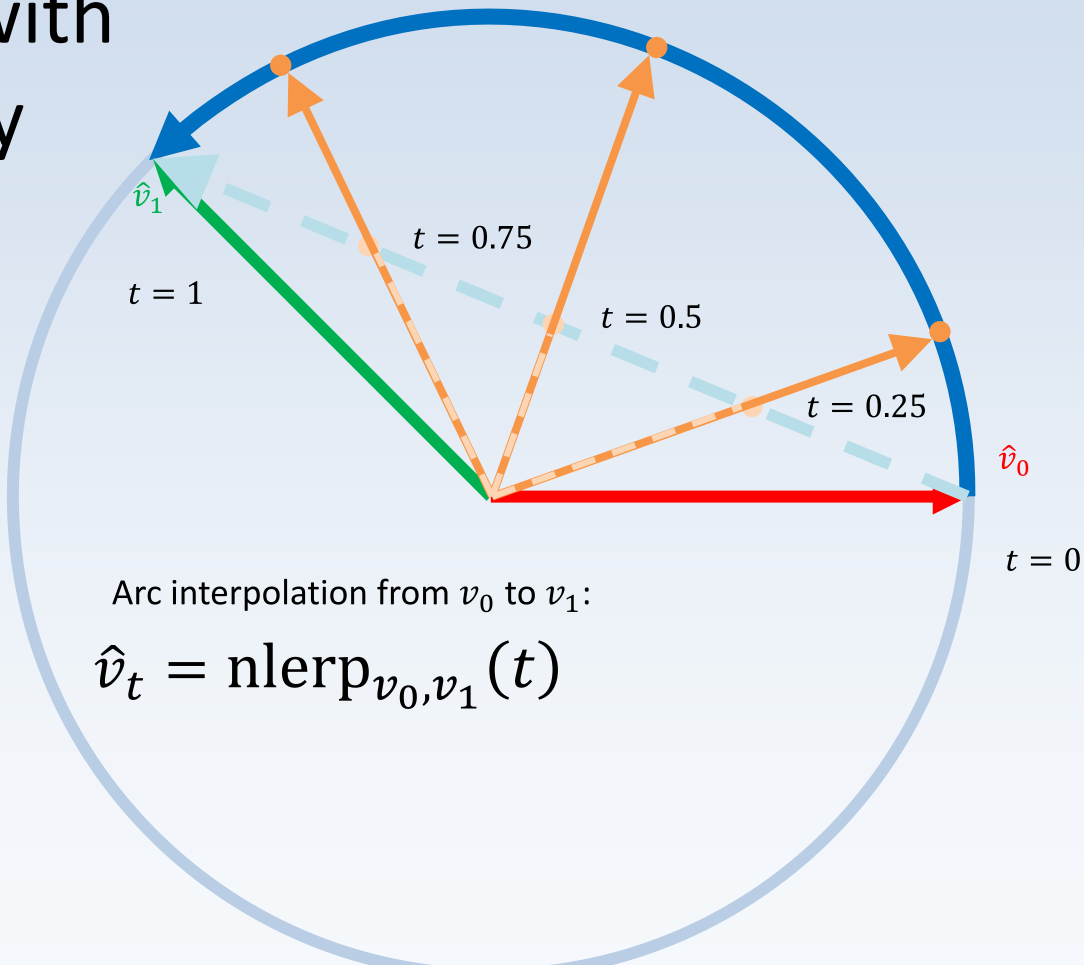
- "*Simulating*" the arc:
- The fast way to compute interpolation along an arc is called ***NLERP***
- "***Normalized Linear Interpolation***"

$$\mathbf{nlerp}_{\boldsymbol{v_0},\boldsymbol{v_1}}(\boldsymbol{t}) = \mathbf{normalize}\left(\mathbf{lerp}_{\boldsymbol{v_0},\boldsymbol{v_1}}(\boldsymbol{t})\right)$$

where the input vectors $\hat{v}_0$ and $\hat{v}_1$ are normalized and the result is also normalized

NLERP: Normalized LERP

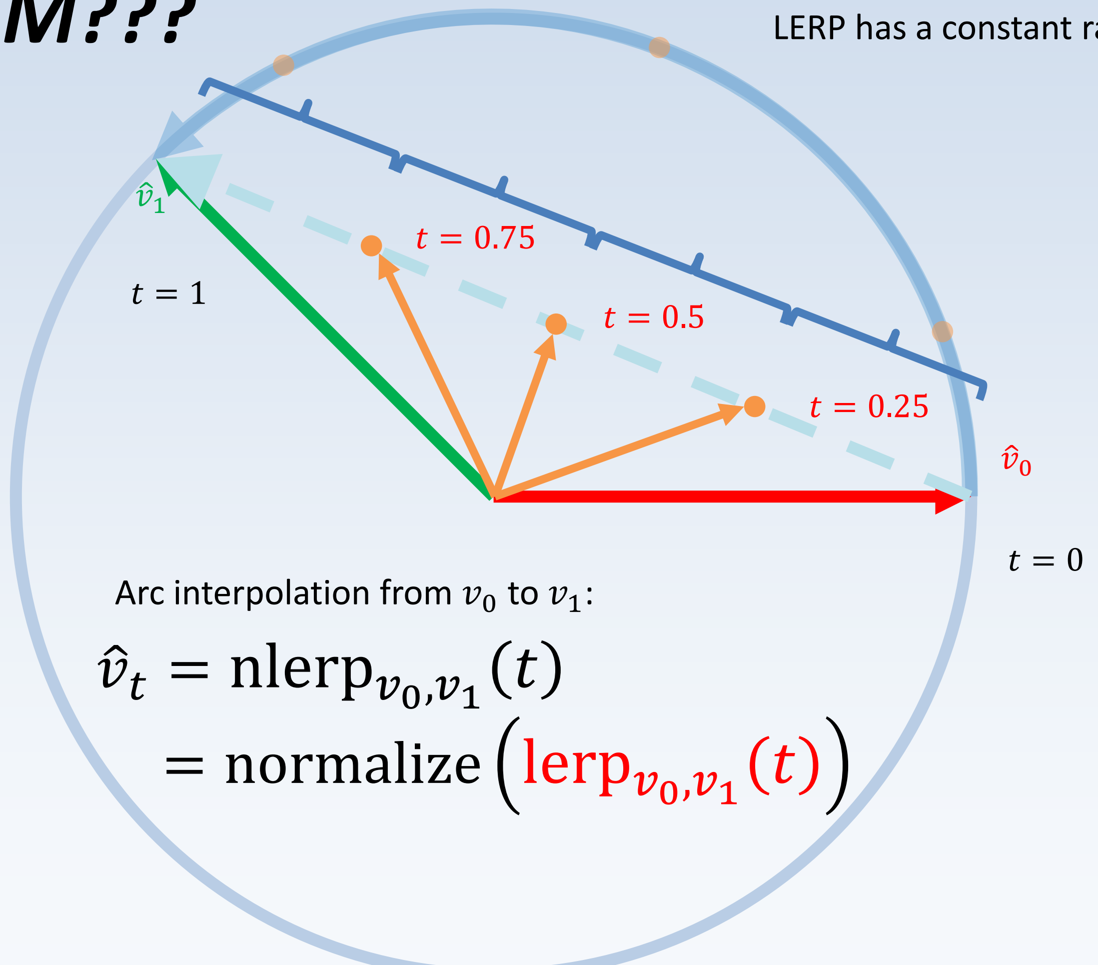
- Works with arbitrary inputs:



NLERP: Normalized LERP

- ***PROBLEM???***

LERP has a constant rate of change!



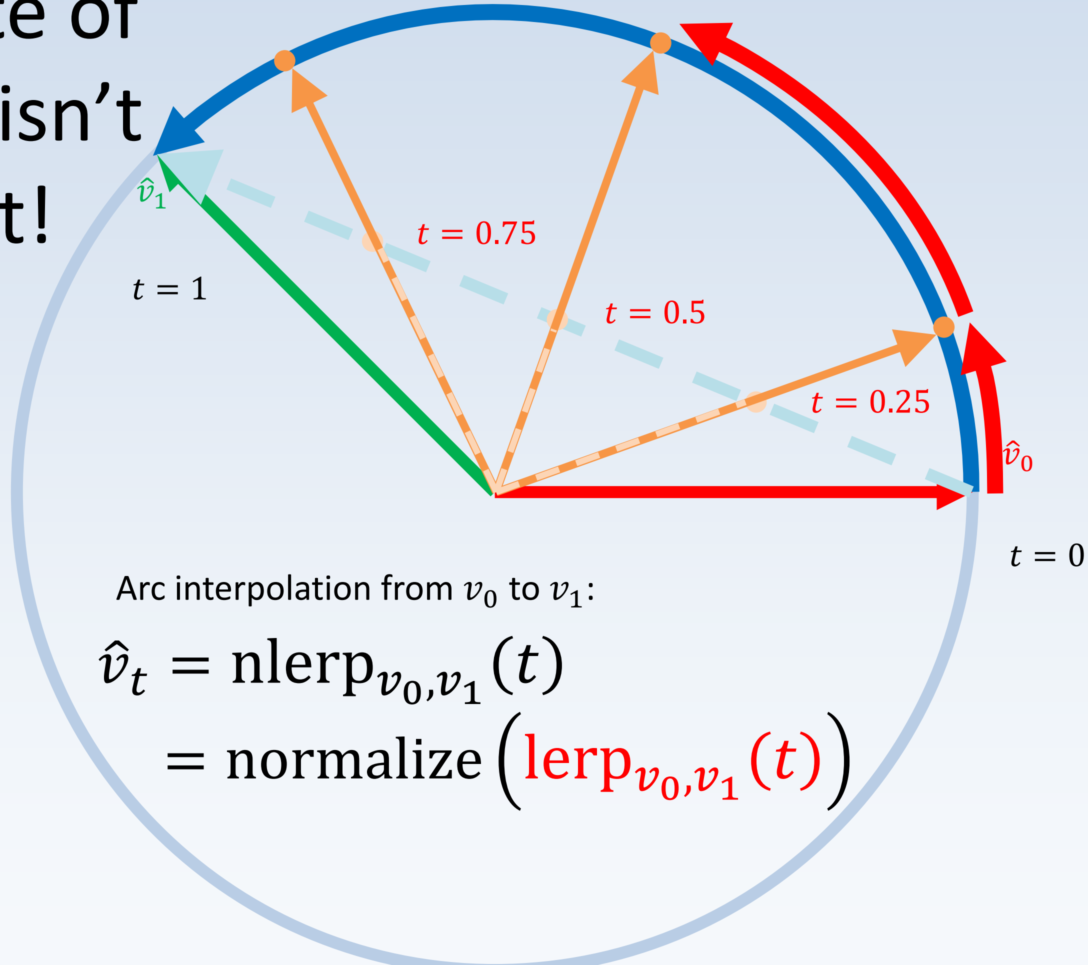
Arc interpolation from v_0 to v_1:

$$\hat{v}_t = \mathrm{nlerp}_{v_0,v_1}(t)$$
$$= \mathrm{normalize}\left(\mathrm{lerp}_{v_0,v_1}(t)\right)$$

NLERP: Normalized LERP

- Arc's rate of change isn't constant!

NLERP does not have a constant rate of change!



Arc interpolation from v_0 to v_1:

$$\hat{v}_t = \mathrm{nlerp}_{v_0,v_1}(t)$$
$$= \mathrm{normalize}\left(\mathrm{lerp}_{v_0,v_1}(t)\right)$$

NLERP: Normalized LERP

- "*Simulating*" the arc:
- NLERP is an efficient way to conform points to a curve… but it has a critical problem
- Using NLERP will yield a path animation that appears *slower* towards the ends and *faster* towards the middle of the arc

Speed = distance / time

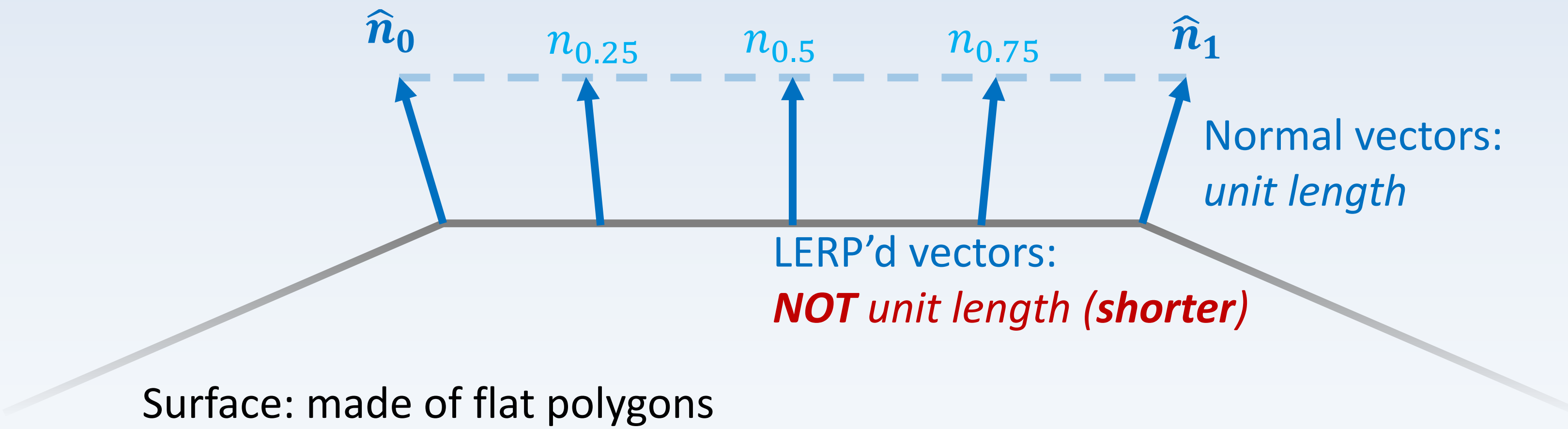
If the distance covered *increases* while time *stays the same*…

… then the speed *increases*!

NLERP: Normalized LERP

- Visible non-spatial example of this anomaly: *per-fragment shading*

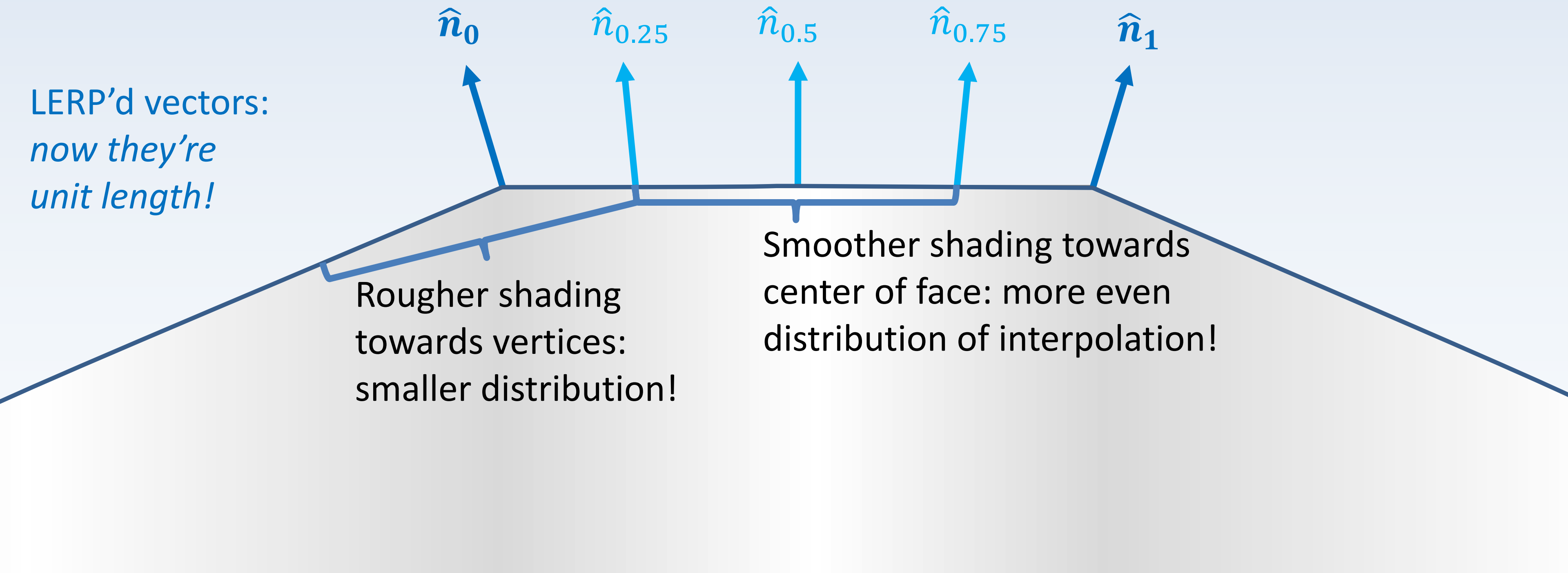
1) Passing *normal* attribute from VS to FS: ***LERP***



NLERP: Normalized LERP

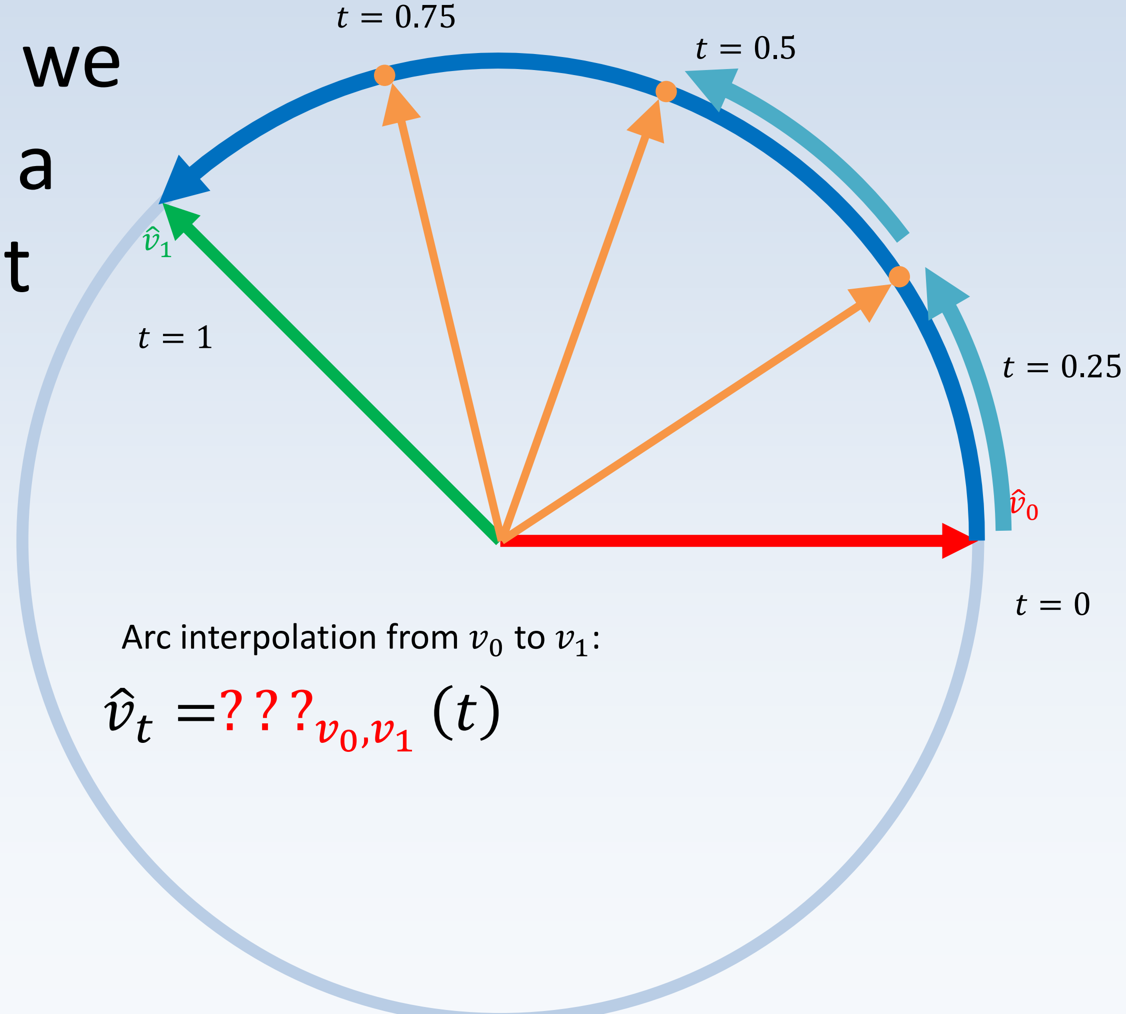
- Visible non-spatial example of this anomaly: *per-fragment shading*

2) Normalizing vector in FS: ***NLERP***



NLERP: Normalized LERP

- How do we achieve a constant speed on the arc???



Arc interpolation from v_0 to v_1:

$$\hat{v}_t = ???_{v_0,v_1}(t)$$

SLERP: Spherical LERP

- There is a more precise arc interpolation algorithm called ***SLERP***
- How it works:
- Instead of linearly interpolating the *points themselves*…
- …we interpolate the *angle separating them*!!!
- Trigonometry helps us represent the result as a point again!

SLERP: Spherical LERP

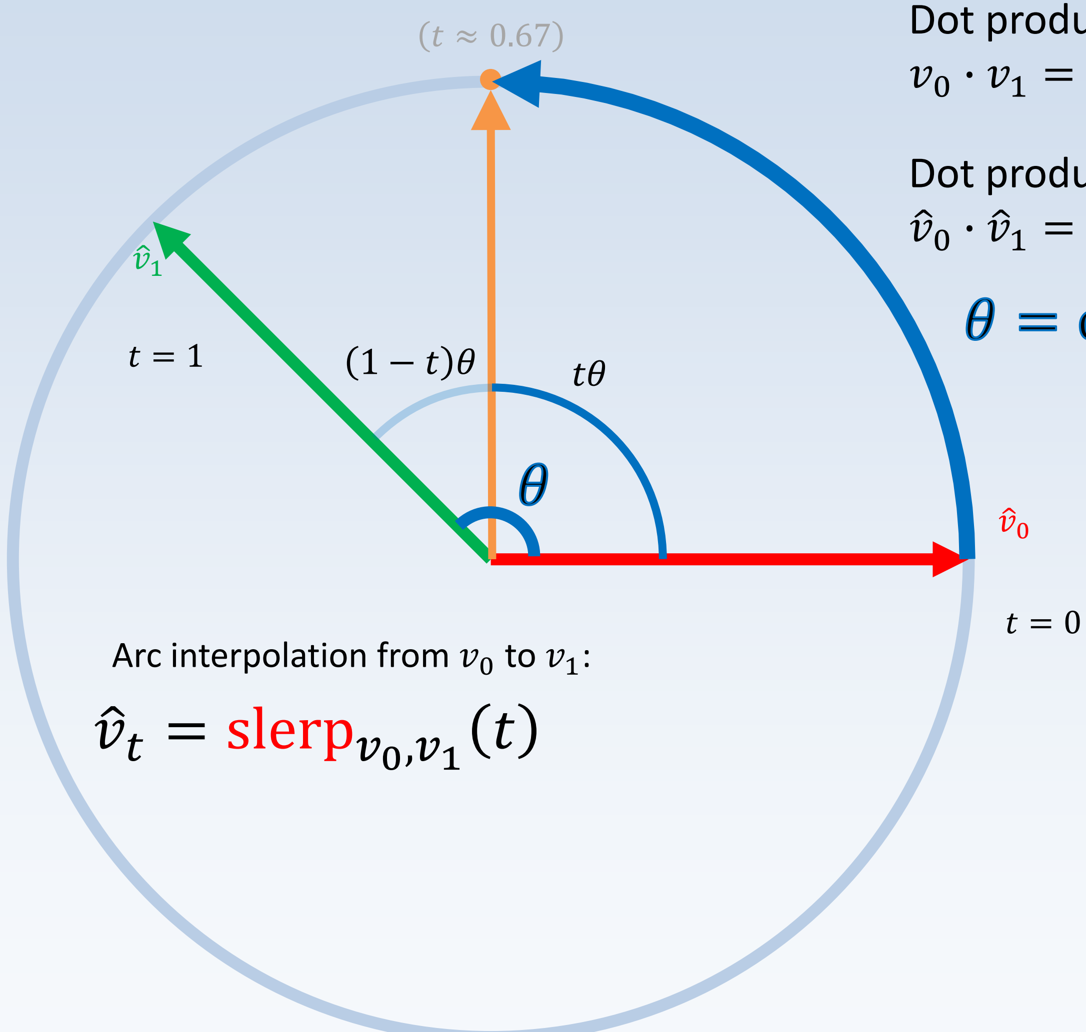
- "***Spherical Linear Interpolation***" (*SLERP*)
- The formula:

$$\mathbf{slerp}_{\boldsymbol{v_0},\boldsymbol{v_1}}(\boldsymbol{t}) = \frac{\mathbf{sin}[(\mathbf{1}-\boldsymbol{t})\boldsymbol{\theta}]\,\boldsymbol{v_0} + \mathbf{sin}[\boldsymbol{t\theta}]\,\boldsymbol{v_1}}{\mathbf{sin}\,\boldsymbol{\theta}}$$

v_0 is the initial value/point/vector (etc.),
v_1 is the goal,
θ is the angle separating the points (see next slide), and
t is our familiar interpolation parameter!

SLERP: Spherical LERP

- SLERP:



Arc interpolation from v_0 to v_1:

$$\hat{v}_t = \text{slerp}_{v_0,v_1}(t)$$

Dot product:

$$v_0 \cdot v_1 = \|v_0\|\|v_1\| \cos\theta$$

Dot product (normalized):

$$\hat{v}_0 \cdot \hat{v}_1 = \cos\theta$$

$$\theta = \cos^{-1}(\hat{v}_0 \cdot \hat{v}_1)$$

SLERP: Spherical LERP

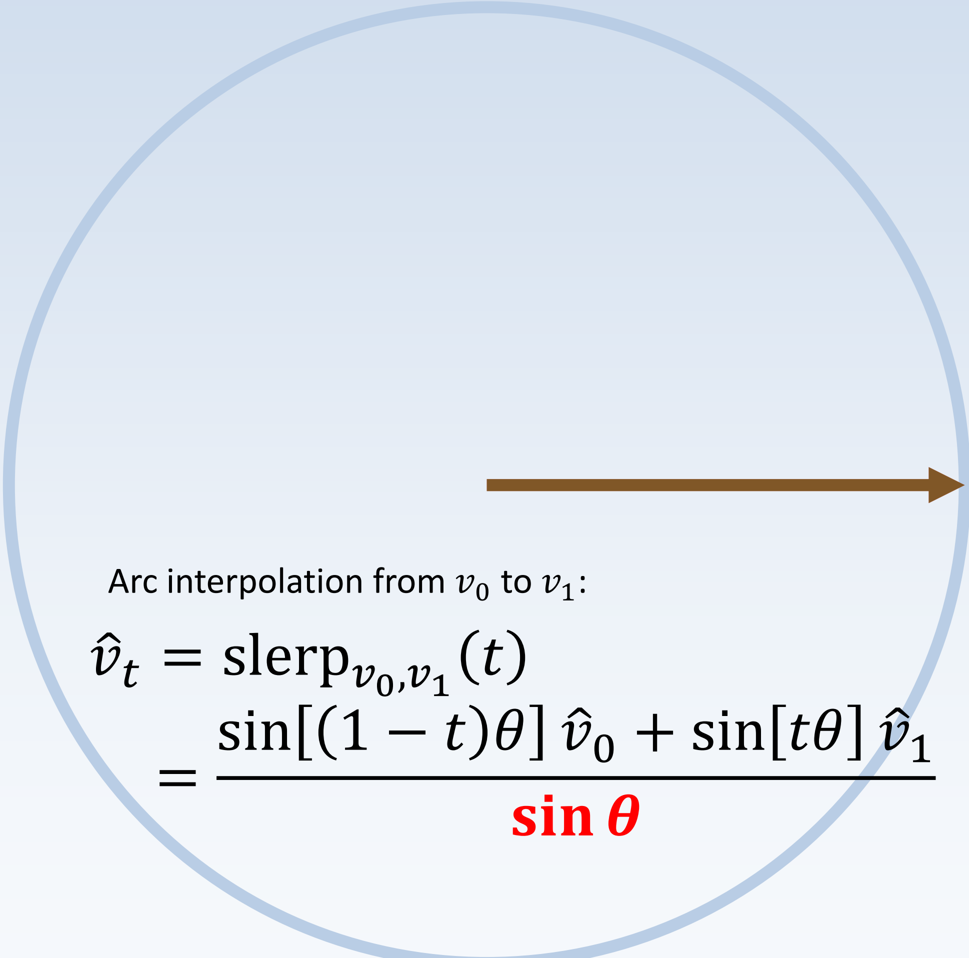
- SLERP gives us the most precise arc interpolation… why???
- We are interpolating the *angle* between the two points instead of the actual *distance*

- That being said… what potential problem might we encounter???

SLERP: Spherical LERP

- Parallel inputs:

$$\hat{v}_0 \cdot \hat{v}_1 = \cos\theta$$
$$\hat{v}_0 \cdot \hat{v}_1 = 1$$
$$\cos\theta = 1$$
$$\theta = 0°$$



Arc interpolation from v_0 to v_1:

$$\hat{v}_t = \mathrm{slerp}_{v_0,v_1}(t)$$
$$= \frac{\sin[(1-t)\theta]\,\hat{v}_0 + \sin[t\theta]\,\hat{v}_1}{\boldsymbol{\sin\theta}}$$

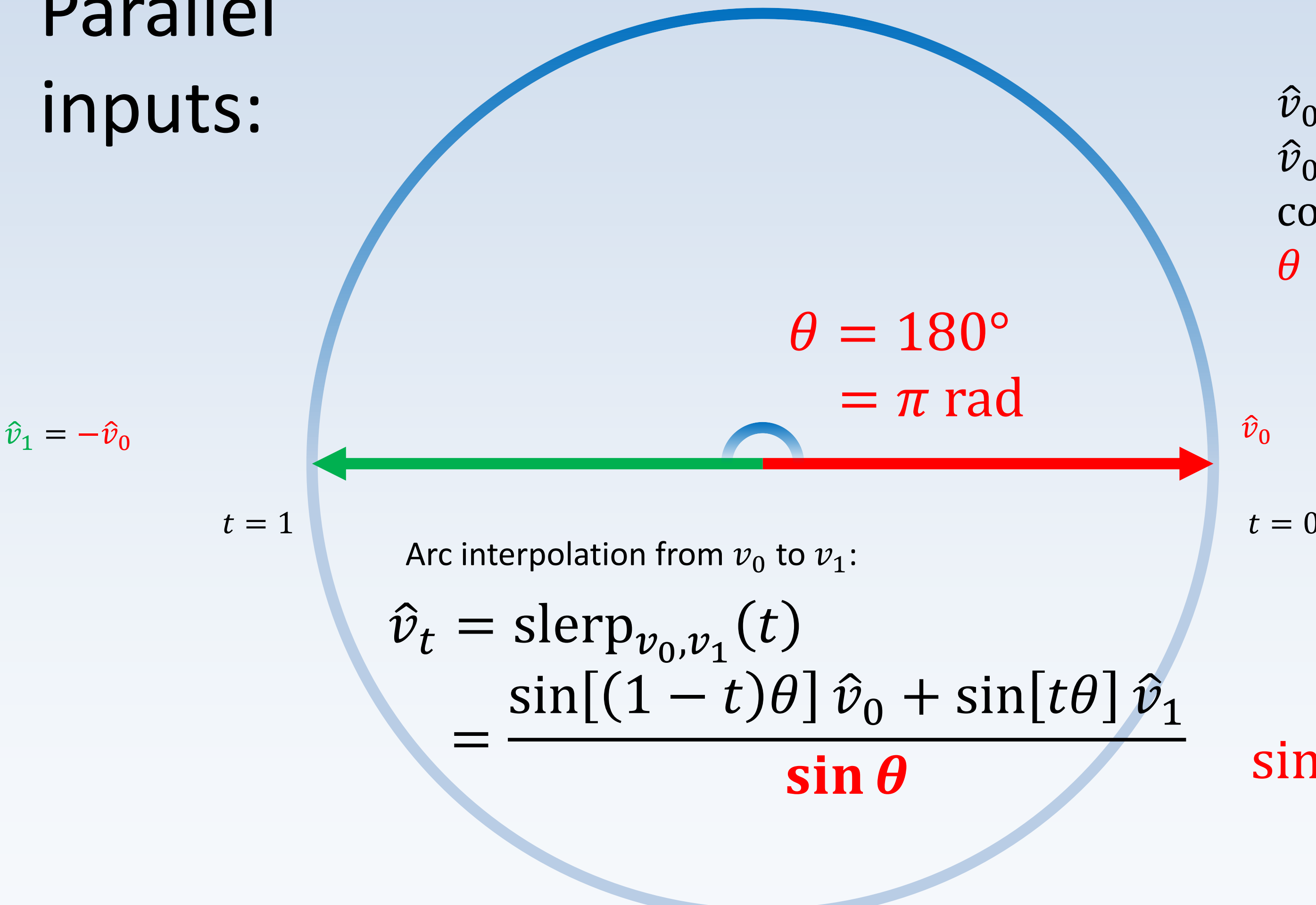
$$\theta = 0°$$

$$\sin 0 = 0$$

SLERP: Spherical LERP

- Parallel inputs:

$$\hat{v}_0 \cdot \hat{v}_1 = \cos\theta$$
$$\hat{v}_0 \cdot \hat{v}_1 = -1$$
$$\cos\theta = -1$$
$$\theta = 180°$$



Arc interpolation from v_0 to v_1:

$$\hat{v}_t = \mathrm{slerp}_{v_0,v_1}(t) = \frac{\sin[(1-t)\theta]\,\hat{v}_0 + \sin[t\theta]\,\hat{v}_1}{\boldsymbol{\sin\theta}}$$

$$\sin\pi = 0$$

SLERP: Spherical LERP

- When the inputs are aligned or parallel, SLERP yields ***division by zero***!!!
- There is no mathematical solution to arc interpolation when this is the case!

- However… we can add some safeguards to our SLERP algorithm…

SLERP: Spherical LERP

- SLERP (v0, v1, t):
- If angle = 0 (or cosine [or dot] = 1),

  result = v0
- Else if angle = 180 (or cosine [or dot] = -1),

  result = LERP (v0, v1, t)
- Else

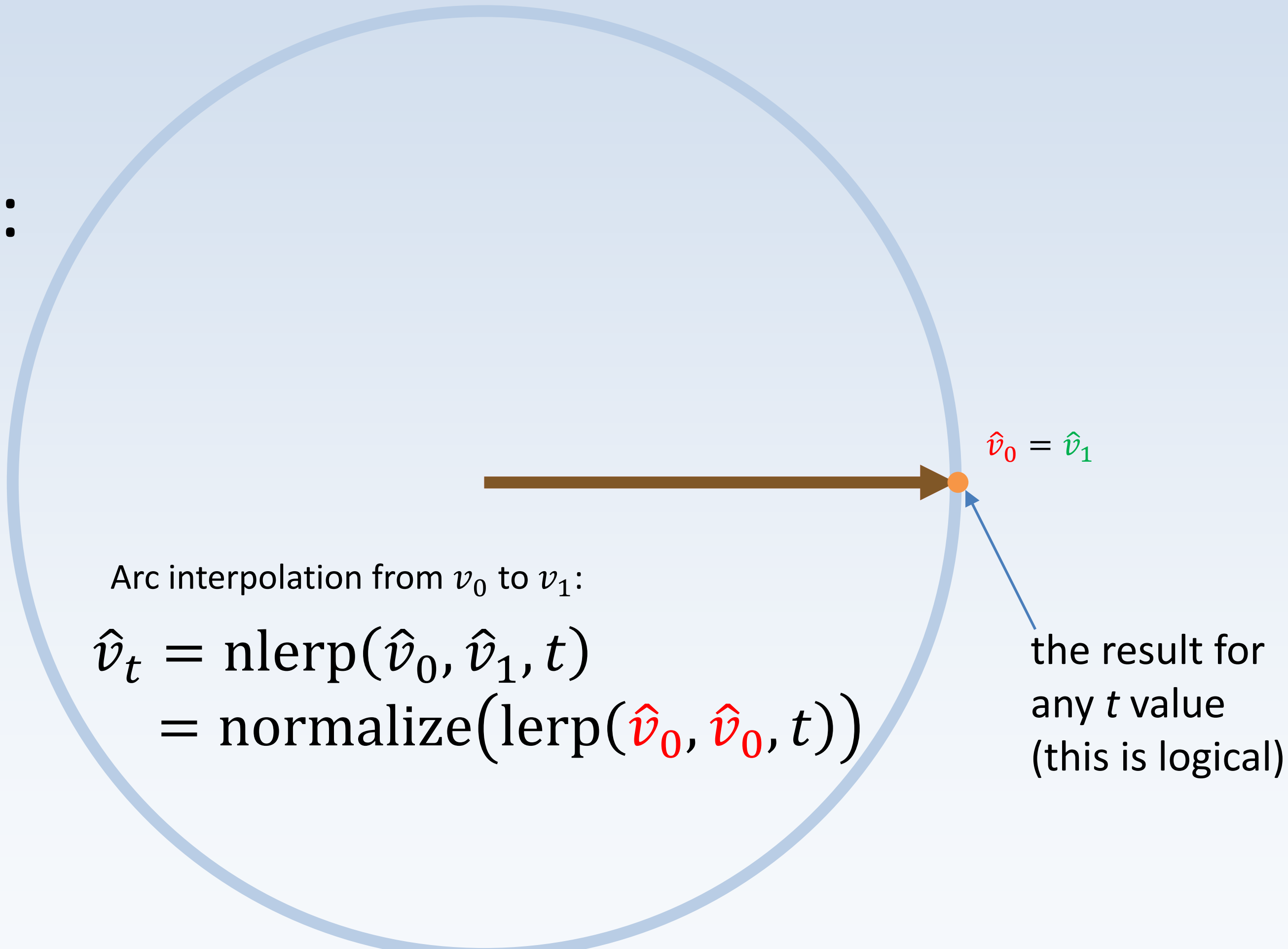
  result = SLERP formula

Problems with Arc Interpolation

- Unfortunately the parallel inputs problem is also a problem for NLERP ☹
- Remember that NLERP only *simulates* an arc
- Since the inputs are parallel, the *displacement vector* between them will also be parallel!
- …we will just end up normalizing to *v0* or *v1* depending on what our *t* value is!

Problems with Arc Interpolation

- Parallel inputs (NLERP):

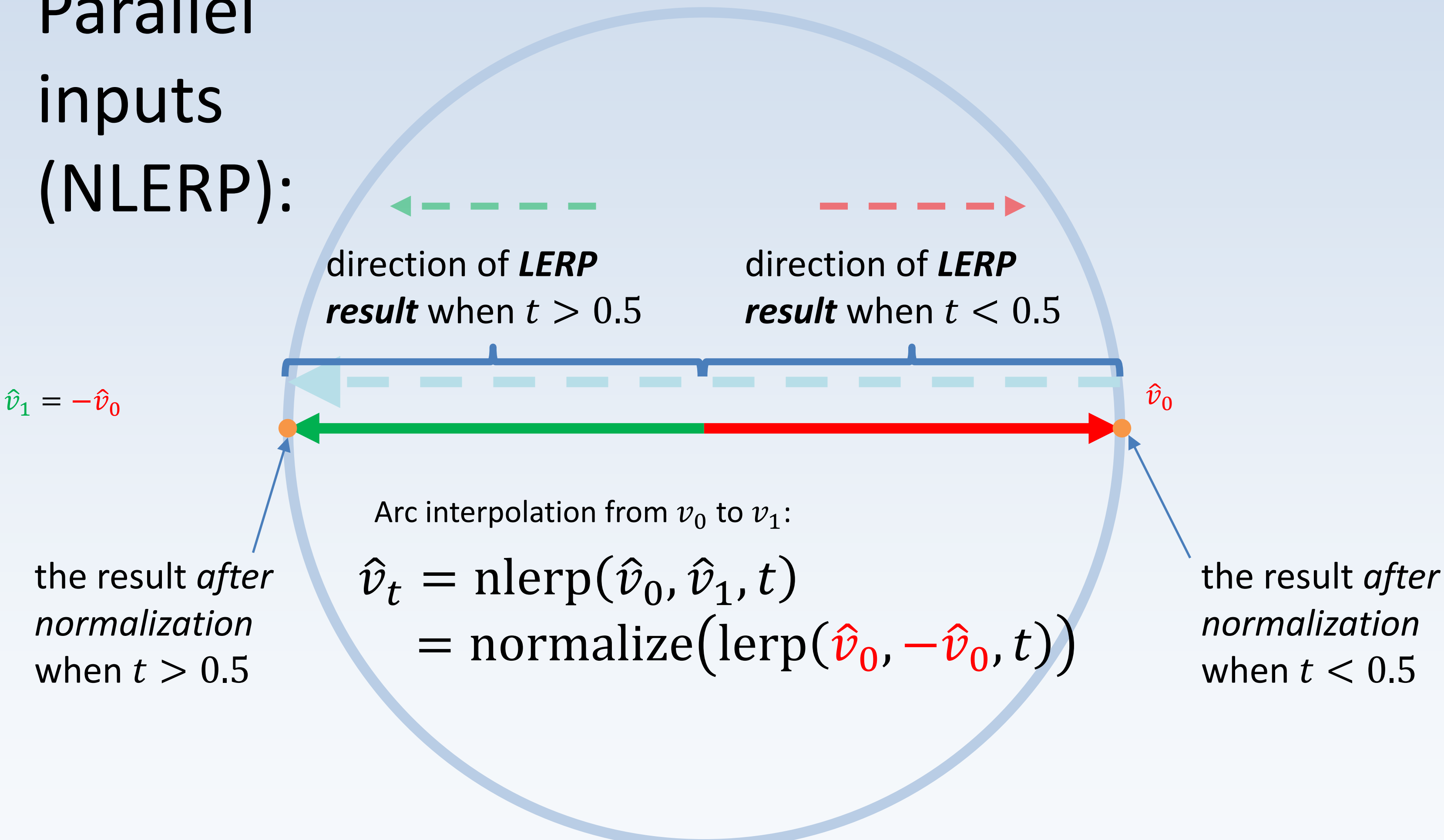


Arc interpolation from v_0 to v_1:

$$\hat{v}_t = \mathrm{nlerp}(\hat{v}_0, \hat{v}_1, t)$$
$$= \mathrm{normalize}\big(\mathrm{lerp}(\hat{v}_0, \hat{v}_0, t)\big)$$

Problems with Arc Interpolation

- Parallel inputs (NLERP):

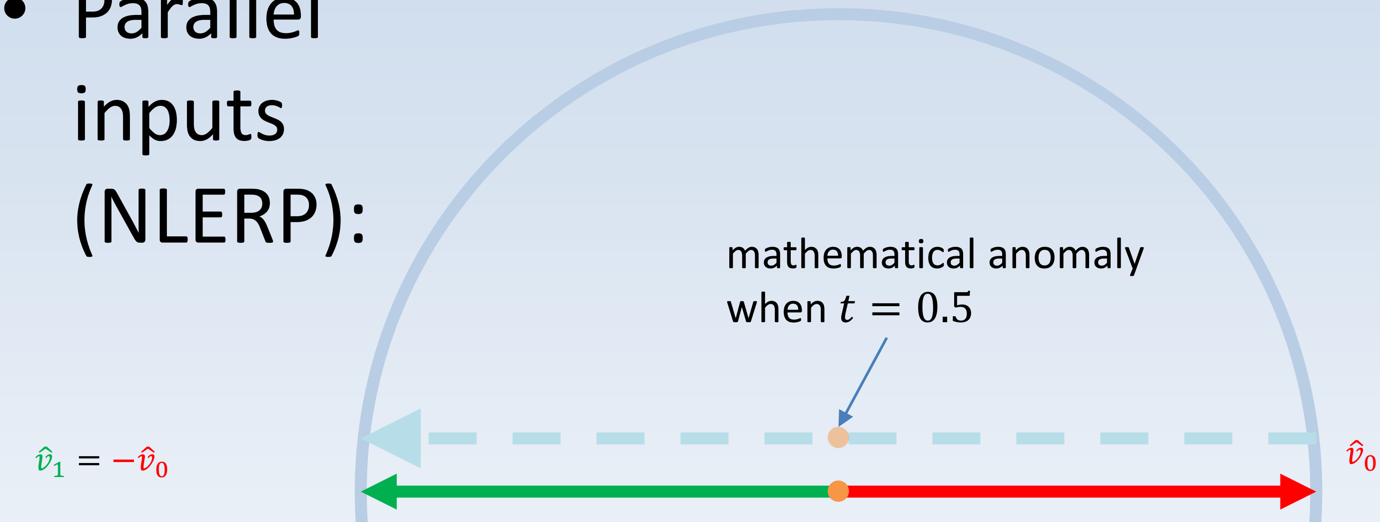


Arc interpolation from v_0 to v_1:

$$\begin{aligned}\hat{v}_t &= \mathrm{nlerp}(\hat{v}_0, \hat{v}_1, t)\\ &= \mathrm{normalize}\big(\mathrm{lerp}(\hat{v}_0, -\hat{v}_0, t)\big)\end{aligned}$$

Problems with Arc Interpolation

- Parallel inputs (NLERP):



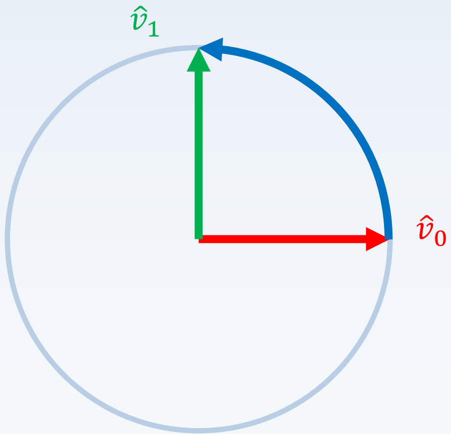
Arc interpolation from v_0 to v_1:

$$
\begin{aligned}
\hat{v}_{0.5} &= \text{nlerp}(\hat{v}_0, \hat{v}_1, 0.5) \\
&= \text{normalize}\big(\text{lerp}(\hat{v}_0, -\hat{v}_0, 0.5)\big) \\
&= \text{normalize}\big((1 - 0.5)\hat{v}_0 - 0.5\hat{v}_0\big) \\
&= \text{normalize}(0.5\hat{v}_0 - 0.5\hat{v}_0) = \mathbf{normalize}\big(\vec{\mathbf{0}}\big)
\end{aligned}
$$

Division by zero again…
need a new safeguard!

Arc Interpolation

- Which algorithm is better? NLERP or SLERP?
- The ubiquitous trade-off in comp. sci.:
- *Performance vs. Precision*
- NLERP is *faster*, but SLERP is *more precise*!



Quaternions

- First described by *William Rowan Hamilton*, an Irish mathematician, in 1843 (published 1865)
- "Vectors are 3D therefore they should represent rotations in 3D… right?"
- No matter how hard he tried, could not figure out how this worked…
- …until one day, while walking across the Brougham Bridge in Ireland…

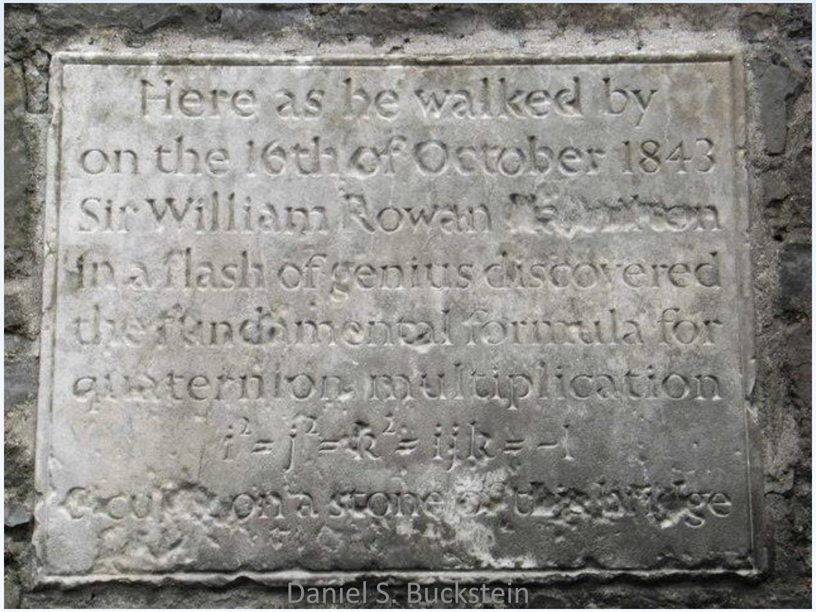
Quaternions

- …Hamilton figured out that a *fourth* component, a *real number*, would be required to control the rotation!
- Four parts → Quaternary
- The concept that frightens many
- …but actually not that scary
- Learning what they are and what they can do will save you in animation/physics

Quaternions

- Hamilton discovered that the basis elements are related by this identity:

$$\boldsymbol{i^2 = j^2 = k^2 = ijk = -1}$$

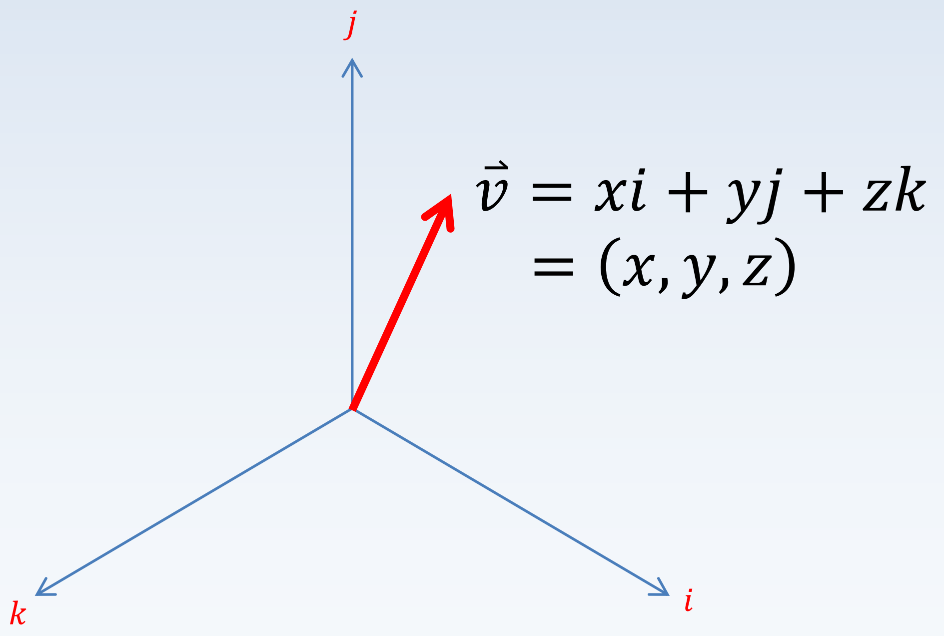


Quaternions

- Vectors are not real numbers, but are rather the sum of three imaginary components:

$$\vec{v} = xi + yj + zk$$

where *x, y,* and *z* are scalars along the respective *basis elements i, j* and *k*



Quaternions

- Adding a fourth basis element, a *real number*, gives us a quaternion:

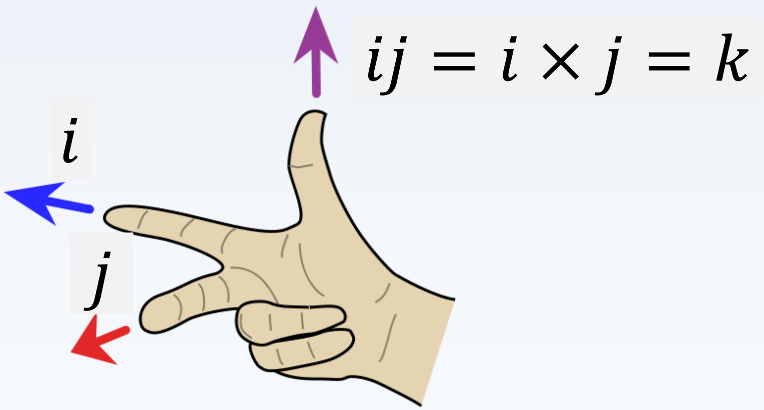
$$q = w(1) + xi + yj + zk$$

where 1, *i*, *j*, and *k* are the basis elements and *w*, *x*, *y*, and *z* are the respective scalars

Quaternions

- The basis elements *i, j* and *k* are related to each other, too:

$$\boldsymbol{ij} = \boldsymbol{k}, \qquad \boldsymbol{ji} = -\boldsymbol{k}$$
$$\boldsymbol{jk} = \boldsymbol{i}, \qquad \boldsymbol{kj} = -\boldsymbol{i}$$
$$\boldsymbol{ki} = \boldsymbol{j}, \qquad \boldsymbol{ik} = -\boldsymbol{j}$$



Quaternions

- Expressed mathematically, a quaternion is the *sum of a scalar and a vector*:

$$q = w + xi + yj + zk$$
$$\vec{v} = 0 + xi + yj + zk$$

Therefore,

$$q = w + \vec{v} = (w, \vec{v}) = (w, x, y, z)$$

Quaternions

- A quaternion with no real part is just a vector:

$$q = 0 + xi + yj + zk = (0, x, y, z) = (0, \vec{v})$$

- This is called a "*pure quaternion*"

- …it's just a vector.

Quaternions

- For all intents and purposes, quaternions share the same main functionalities of vectors… but in 4 dimensions:
- Dot product: $q_0 \cdot q_1 = w_0 w_1 + x_0 x_1 + y_0 y_1 + z_0 z_1$
- Magnitude: $\|q\| = \sqrt{w^2 + x^2 + y^2 + z^2}$, $\|q\|^2 = q \cdot q$
- Normalize: $\hat{q} = \frac{q}{\|q\|}$

Quaternions

- We are concerned with quaternions that have a length of *one* (normalized)
- Normalized quaternions represent *rotations*!
  - Called “versors” in pure math terms
- Rotation quaternions have huge advantages:
- Directly translates to axis-angle form
- No Euler angles, no gimbal lock
- Each quaternion maps to exactly *one* rotation!

Quaternions

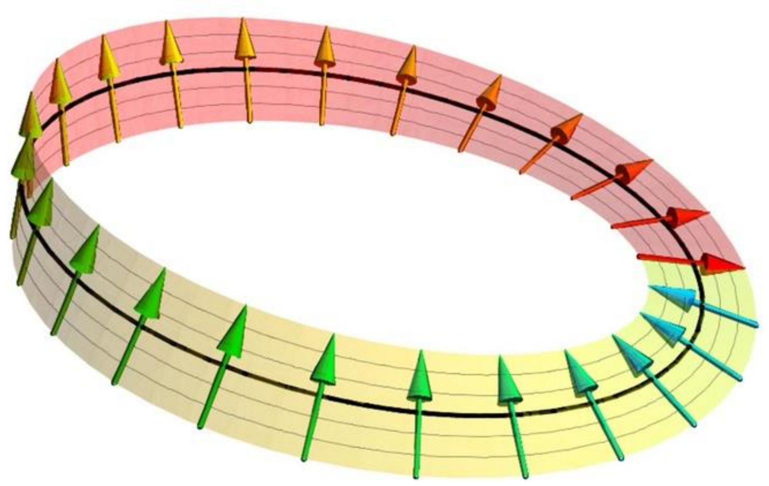
- ***Converting axis-angle to quaternion***:
- Given some angle θ and some normalized axis $\hat{n}$, a rotation quaternion is synthesized as:

$$w = \cos\left(\frac{\theta}{2}\right), \qquad \vec{v} = \hat{n}\sin\left(\frac{\theta}{2}\right)$$

$$\hat{q} = (w, \vec{v}) = \left(\cos\left(\frac{\theta}{2}\right), \hat{n}\sin\left(\frac{\theta}{2}\right)\right)$$

Quaternions

- Wait… why the *half*-angle???
- The behavior of a quaternion rotation forms a *spinor*, or *Mobius strip*:
- A full rotation is 720° for a spinor
- Half angle is the 3D equivalent



Quaternions

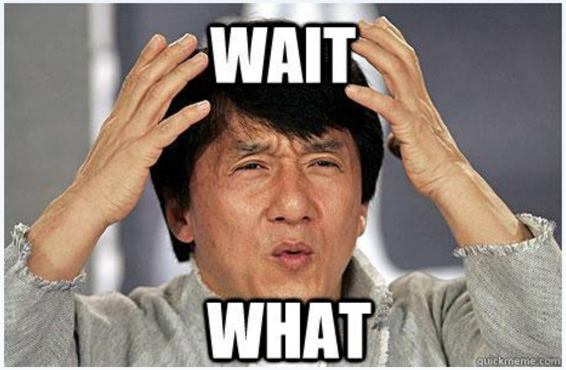
- The identity quaternion (no rotation) is
$$\hat{q} = \left(1, \vec{0}\right) = (1, 0, 0, 0)$$
- This is what you get from a $0°$ rotation, regardless of the axis:
$$\hat{q} = \left(\cos\left(\frac{0}{2}\right), \hat{n}\sin\left(\frac{0}{2}\right)\right) = (\cos 0\,, \hat{n}\sin 0)$$
$$= (1, 0\hat{n}) = (1, 0, 0, 0)$$

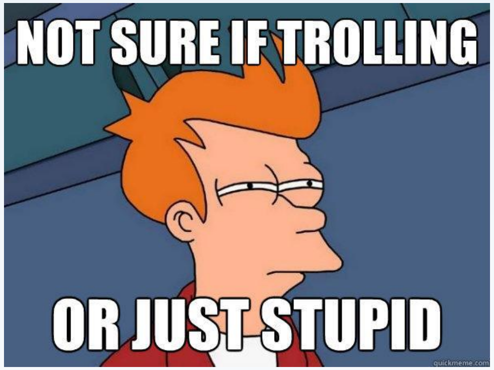
Quaternions

- What about a $360°$ (2π rad) rotation about any axis? Should be the same, right???

$$\hat{q} = \left(\cos\left(\frac{2\pi}{2}\right), \hat{n}\sin\left(\frac{2\pi}{2}\right)\right) = (\cos\pi, \hat{n}\sin\pi)$$

$$= (-1, 0\hat{n}) = (-1, 0, 0, 0)$$





Quaternions

- In theory, a $360°$ rotation is the same as a $0°$ rotation… but because of the *spinor* shape…
- …it would take a full $720°$ rotation to return to the "original" orientation
- Rotation quaternions have a special property:

$$\widehat{\boldsymbol{q}} \equiv -\widehat{\boldsymbol{q}}$$

which means that a quaternion *and its negative* have the exact same *meaning*!

(Triple-bar equals sign is ***logical equivalence***: not *equal*, but do the same thing)

Quaternions

- So if the negative rotation quaternion represents the exact same rotation…
- …how do we determine the *inverse*???
- ***Conjugate***:

For any quaternion

$$q = (w, \vec{v}) = (w, x, y, z)$$

the *conjugate* is

$$q^* = (w, -\vec{\boldsymbol{v}}) = (w, -\boldsymbol{x}, -\boldsymbol{y}, -\boldsymbol{z})$$

Quaternions

- For any quaternion $q = (w, \vec{v})$, the inverse is

$$q^{-1} = \frac{q^*}{\|q\|^2}$$

which means that for a rotation quaternion (which has a magnitude of 1), the inverse is

$$\hat{q}^{-1} = \hat{q}^*$$

…just like how a rotation matrix's inverse is just the transpose!!!

Quaternions

- Why is the ***conjugate*** the inverse and not the ***negative***???
- We already saw that the *negative* quaternion represents the same rotation…
- If we flip the *axis* while using the same *angle*, the result is the opposite rotation
- Negating the entire quaternion is the same as flipping both the axis *and* the angle (because cosine!)

Quaternions

- How do we extract an axis and an angle from a quaternion???

$$w = \cos\left(\frac{\theta}{2}\right) \quad \rightarrow \quad \theta = 2\cos^{-1}(w)$$

$$\vec{v} = \hat{n}\sin\left(\frac{\theta}{2}\right) \quad \rightarrow \quad \hat{n} = \frac{1}{\sin\left(\frac{\theta}{2}\right)}\vec{v} \text{ or } \hat{n} = \frac{\vec{v}}{|\vec{v}|}$$

Quaternions

- **Concatenation (multiplication)**:
- The long way: take the product of two mathematical quaternions

$$q_0 = (w_0, \vec{v}_0) = (w_0, x_0, y_0, z_0)$$
$$= w_0 + x_0 i + y_0 j + z_0 k$$

$$q_1 = (w_1, \vec{v}_1) = (w_1, x_1, y_1, z_1)$$
$$= w_1 + x_1 i + y_1 j + z_1 k$$

Quaternions

- **Concatenation (full expansion)**:

$$q_0q_1 = (w_0 + x_0 i + y_0 j + z_0 k)(w_1 + x_1 i + y_1 j + z_1 k)$$

$$\begin{aligned} &= w_0w_1 + w_0x_1 i + w_0y_1 j + w_0z_1 k \\ &+x_0w_1 i + x_0 i x_1 i + x_0 i y_1 j + x_0 i z_1 k \\ &+y_0w_1 j + y_0 j x_1 i + y_0 j y_1 j + y_0 j z_1 k \\ &+z_0w_1 k + z_0 k x_1 i + z_0 k y_1 j + z_0 k z_1 k \end{aligned}$$

Quaternions

- **Concatenation**:

$$q_0q_1 = (w_0 + x_0 i + y_0 j + z_0 k)(w_1 + x_1 i + y_1 j + z_1 k)$$

$$\begin{aligned} &= w_0w_1 + w_0x_1 i + w_0y_1 j + w_0z_1 k \\ &\quad + x_0w_1 i + x_0x_1 i^2 + x_0y_1 ij + x_0z_1 ik \\ &\quad + y_0w_1 j + y_0x_1 ji + y_0y_1 j^2 + y_0z_1 jk \\ &\quad + z_0w_1 k + z_0x_1 ki + z_0y_1 kj + z_0z_1 k^2 \end{aligned}$$

$ij = k, \quad ji = -k$

$jk = i, \quad kj = -i$

$ki = j, \quad ik = -j$

$i^2 = j^2 = k^2 = ijk = -1$

Quaternions

- **Concatenation**:

$$q_0 q_1 = (w_0 + x_0 i + y_0 j + z_0 k)(w_1 + x_1 i + y_1 j + z_1 k)$$

$$\begin{aligned} &= w_0 w_1 + w_0 x_1 i + w_0 y_1 j + w_0 z_1 k \\ &\quad + x_0 w_1 i - x_0 x_1 + x_0 y_1 k - x_0 z_1 j \\ &\quad + y_0 w_1 j - y_0 x_1 k - y_0 y_1 + y_0 z_1 i \\ &\quad + z_0 w_1 k + z_0 x_1 j - z_0 y_1 i - z_0 z_1 \end{aligned}$$

$ij = k,\quad ji = -k$

$jk = i,\quad kj = -i$

$ki = j,\quad ik = -j$

$i^2 = j^2 = k^2 = ijk = -1$

Quaternions

- **Concatenation**:

$$q_0 q_1 = (w_0 + x_0 i + y_0 j + z_0 k)(w_1 + x_1 i + y_1 j + z_1 k)$$

$$\begin{aligned}= &(w_0 w_1 - x_0 x_1 - y_0 y_1 - z_0 z_1)1 \\ &+(w_0 x_1 + x_0 w_1 + y_0 z_1 - z_0 y_1)i \\ &+(w_0 y_1 - x_0 z_1 + y_0 w_1 + z_0 x_1)j \\ &+(w_0 z_1 + x_0 y_1 - y_0 x_1 + z_0 w_1)k\end{aligned}$$

Quaternions

- **Concatenation**:
- Luckily there is a compact formula… so don't panic about all that mess:

$$q_0 = (w_0, \vec{v}_0), \qquad q_1 = (w_1, \vec{v}_1)$$

$$\boldsymbol{q_0 q_1} = \begin{pmatrix} \boldsymbol{w_0 w_1 - \vec{v}_0 \cdot \vec{v}_1} \;, \\ \boldsymbol{w_0 \vec{v}_1 + w_1 \vec{v}_0 + \vec{v}_0 \times \vec{v}_1} \end{pmatrix} \begin{matrix} \leftarrow \text{real part} \\ \leftarrow \text{vector part} \end{matrix}$$

Quaternions

- **Concatenation**:
- Like with matrices, concatenation is ***non-commutative***: $q_0 q_1 \neq q_1 q_0$
- Like with matrices (again), concatenation is ***associative***: $(q_0 q_1) q_2 = q_0 (q_1 q_2)$
- Like with matrices (yet again), the order of operation is ***right to left***: e.g. with $q = q_0 q_1$, *q1* happens first!

Quaternions

- **Rotating a vector**:
- Applying a rotation to a single vector using a quaternion is not as simple as it is with rotations…

$$\vec{v}' = R\vec{v} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Quaternions

- **Rotating a vector**:
- 4-dimensional math wizardry must be used
- First represent the vector as a *pure quaternion*: $\vec{v} = (0, x, y, z)$
- Then plug it into this formula (the result will also be a pure quaternion):

$$\vec{v}' = \hat{q}\vec{v}\hat{q}^*$$

Quaternions

- **Rotating a vector**:
- As always, there is a better formula:
- Let $\vec{v}$ represent the vector we want to rotate, and let $\vec{r}$ represent the quaternion's vector component:

Quaternion $\hat{q} = (w, \vec{r})$ rotating vector $\vec{v}$

$$\boldsymbol{\vec{v}' = \vec{v} + 2\vec{r} \times (\vec{r} \times \vec{v} + w\vec{v})}$$

Quaternions

- **Interpolation**:
- Remember that quaternions are *four-dimensional*
- They represent a *rotation*, which is an *action*… as opposed to a *point*, which is a physical concept
- …how do you interpolate an *action*???

Quaternion SLERP

- The SLERP formula with quaternion inputs:

$$\mathbf{slerp}_{\hat{q}_0,\hat{q}_1}(\boldsymbol{t}) = \frac{\mathbf{sin}[(\mathbf{1}-\boldsymbol{t})\boldsymbol{\Omega}]\,\hat{\boldsymbol{q}}_{\mathbf{0}} + \mathbf{sin}[\boldsymbol{t}\boldsymbol{\Omega}]\,\hat{\boldsymbol{q}}_{\mathbf{1}}}{\mathbf{sin}\,\boldsymbol{\Omega}}$$

$\hat{q}_0$ is the initial rotation,

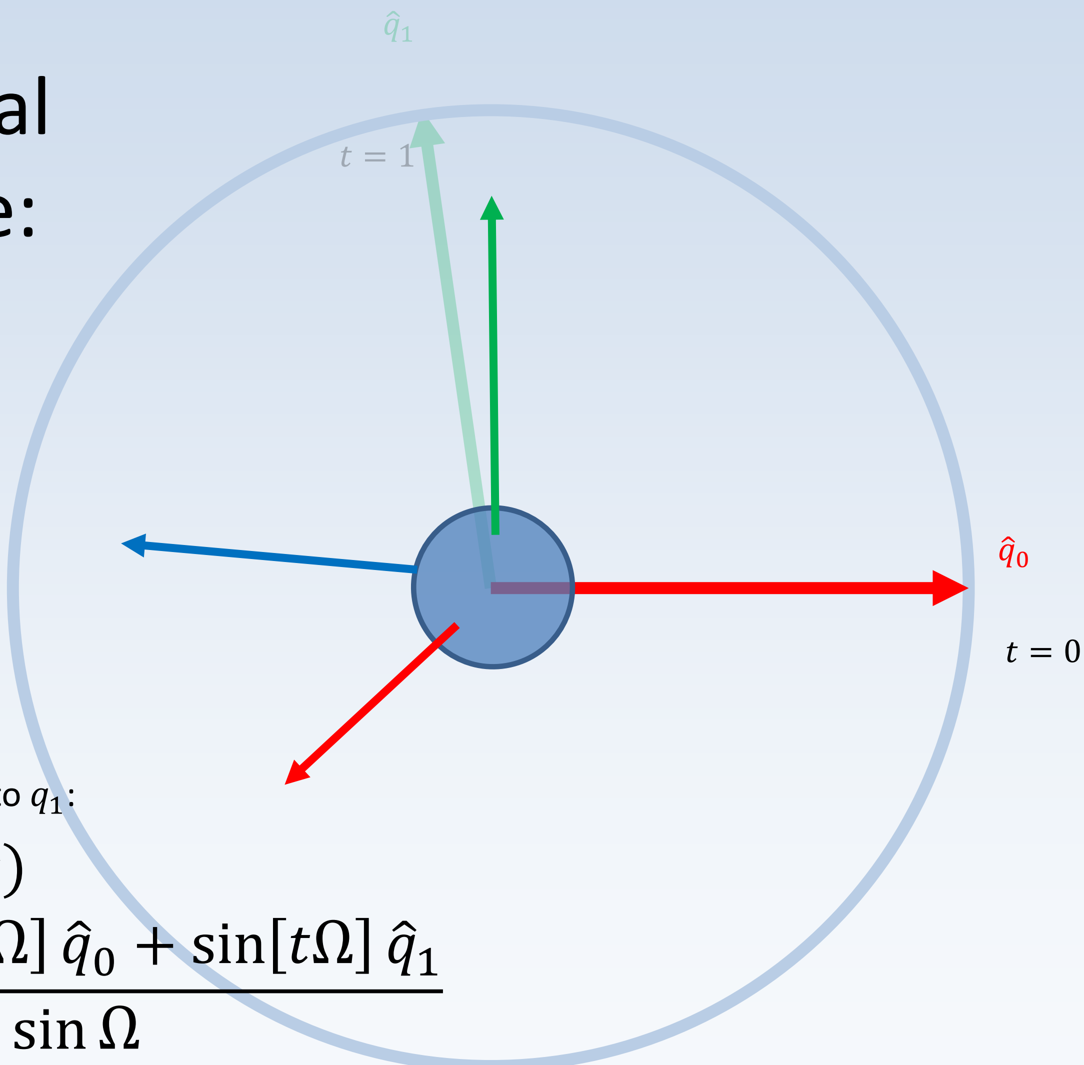
$\hat{q}_1$ is the end rotation,

Ω is the "*angle*" separating the points, and

t is our familiar interpolation parameter!

Quaternion SLERP

- Graphical example:

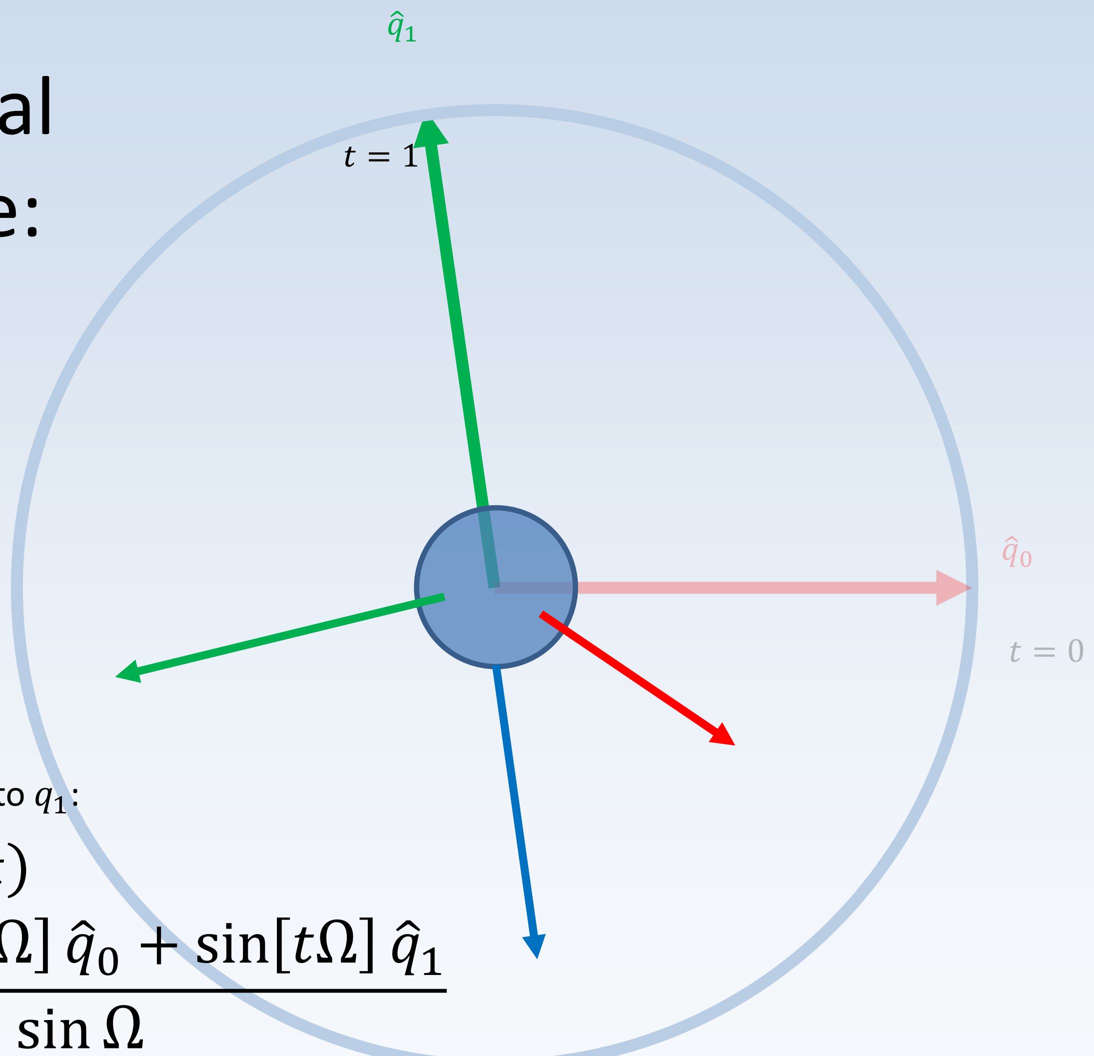


Quaternion SLERP from q_0 to q_1:

$$\hat{q}_t = \mathrm{slerp}_{\hat{q}_0,\hat{q}_1}(t)$$
$$= \frac{\sin[(1-t)\Omega]\,\hat{q}_0 + \sin[t\Omega]\,\hat{q}_1}{\sin\Omega}$$

Quaternion SLERP

- Graphical example:



Quaternion SLERP from q_0 to q_1:

$$\hat{q}_t = \mathrm{slerp}_{\hat{q}_0,\hat{q}_1}(t)$$
$$= \frac{\sin[(1-t)\Omega]\,\hat{q}_0 + \sin[t\Omega]\,\hat{q}_1}{\sin\Omega}$$

Quaternion SLERP

- Why does this work so nicely for quaternions?
- Quaternion SLERP smoothly interpolates axis and angle simultaneously
- Maintains length of 1, so the result is always a normalized quaternion…
- …which is a valid rotation!!!

Quaternion SLERP

- Luckily, quaternion SLERP works the same way in 4 dimensions as in 2 or 3 dimensions
- The "parallel inputs" problem also applies!
- But we can solve it… we have a special optimization for quaternion SLERP because…
- "Parallel" quaternions represent the same rotation!

$$\hat{q} \equiv -\hat{q}$$

Quaternion SLERP

- Parallel inputs:

$$\hat{q}_0 \cdot \hat{q}_1 = \cos\Omega$$
$$\hat{q}_0 \cdot \hat{q}_1 = 1$$
$$\cos\Omega = 1$$
$$\Omega = 0°$$

$\hat{q}_0 = \hat{q}_1$

Quaternion SLERP from q_0 to q_1:

$$\hat{q}_t = \mathrm{slerp}_{\hat{q}_0,\hat{q}_1}(t) = \frac{\sin[(1-t)\Omega]\,\hat{q}_0 + \sin[t\Omega]\,\hat{q}_1}{\mathbf{\sin\Omega}}$$

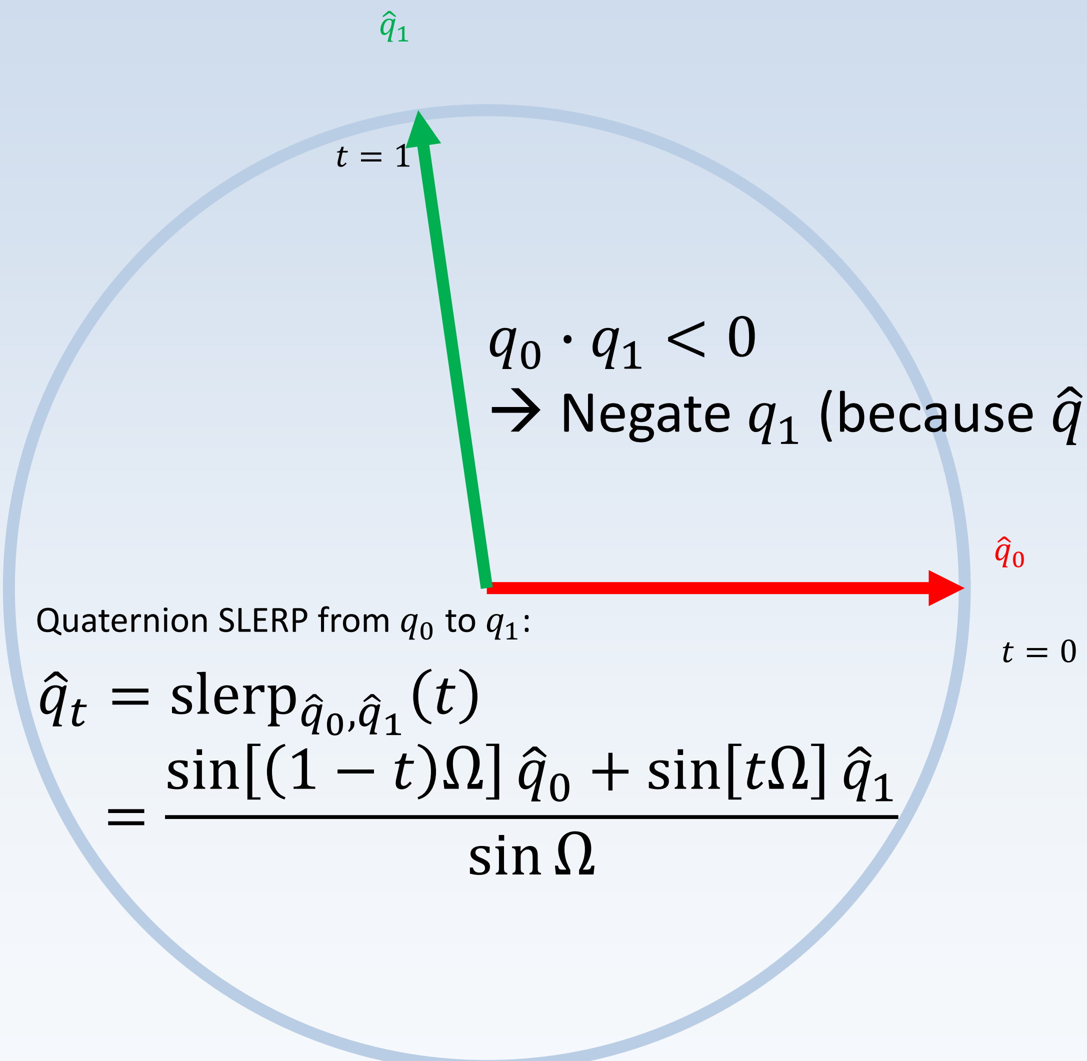
$t = 0$
$t = 1$

$$\Omega = 0°$$

$$\sin 0° = 0$$

Quaternion SLERP

- Parallel inputs:



$q_0 \cdot q_1 < 0$

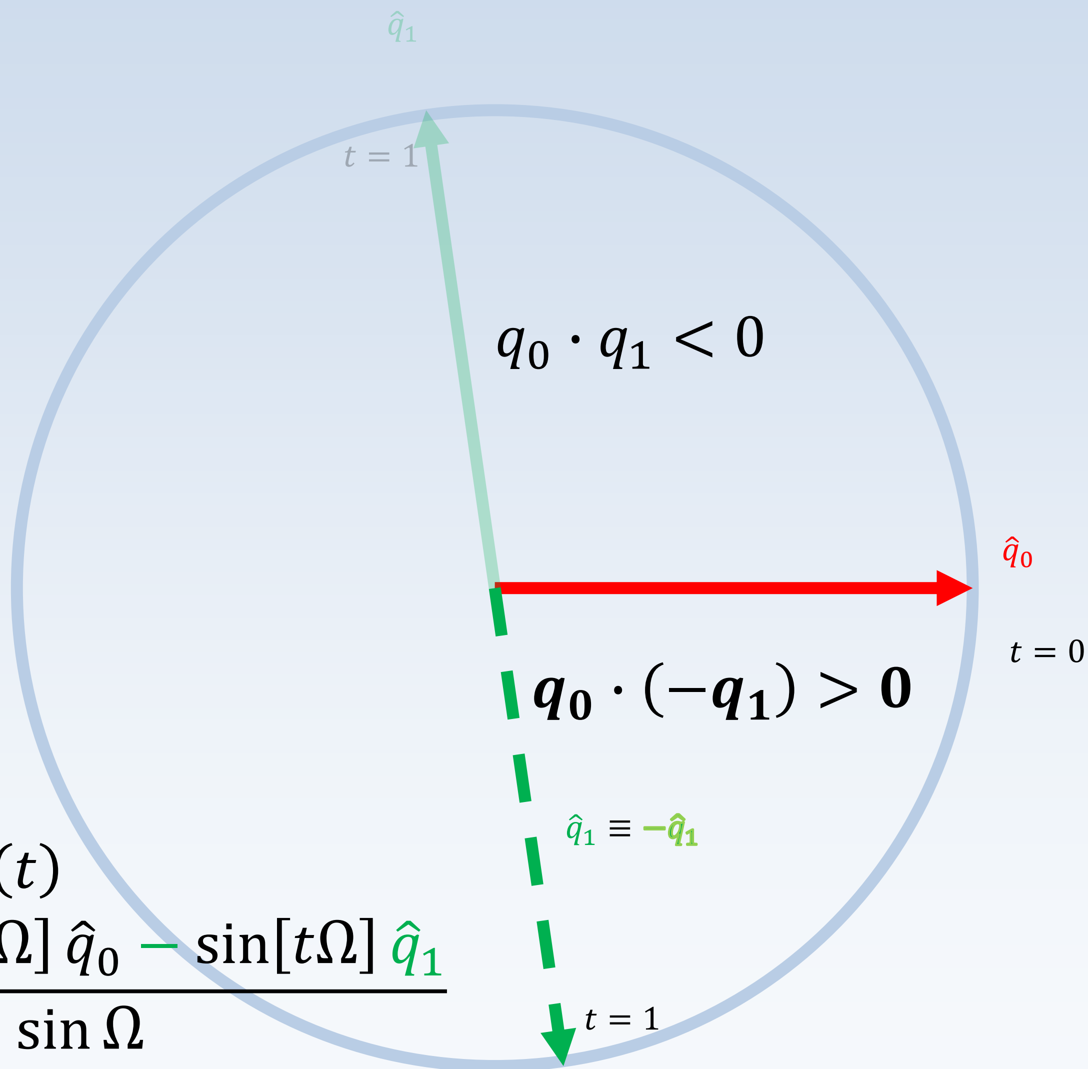
→ Negate q_1 (because $\hat{q} \equiv -\hat{q}$)

Quaternion SLERP from q_0 to q_1:

$$\hat{q}_t = \mathrm{slerp}_{\hat{q}_0,\hat{q}_1}(t)$$
$$= \frac{\sin[(1-t)\Omega]\,\hat{q}_0 + \sin[t\Omega]\,\hat{q}_1}{\sin\Omega}$$

Quaternion SLERP

- Parallel inputs:



$$\hat{q}_t = \mathrm{slerp}_{\hat{q}_0, -\hat{q}_1}(t)$$
$$= \frac{\sin[(1-t)\Omega]\,\hat{q}_0 - \sin[t\Omega]\,\hat{q}_1}{\sin\Omega}$$

Quaternion SLERP

- SLERP(q0, q1, t):
- If cosine [or dot] < 0,

  q1 = -q1 (because they mean the same thing!)

  (also negate dot product to get the proper theta!)
- If cosine [or dot] >= 1

  result = q0
- Proceed with slerp formula as normal ☺

Quaternion SLERP

- ***PRO TIP 4 U***: SLERP is very computationally expensive!!!
- 3 *sin* calls (SLERP) vs. 1 *sqrt* call (NLERP)
- NLERP is a worthwhile alternative if you want to ditch some precision for performance!
- The ubiquitous performance vs. precision dilemma strikes again…

Quaternions vs. Matrices

- **Performance (speed)**:
- Storage requirements:

Rotation matrix: 9 floats

Quaternion: ***4 floats***

$$R = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

$$\boxed{q = (w, x, y, z)}$$

Quaternions vs. Matrices

- **Performance (speed)**:
- Concatenation (chaining operations):

Rotation matrices ($R_0 R_1$): 45

Quaternions ($q_0 q_1$): ***28***

$$R_0 R_1 = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

$$\boxed{q_0 q_1 = \begin{pmatrix} w_0 w_1 - \vec{v}_0 \cdot \vec{v}_1 \ , \\ w_0 \vec{v}_1 + w_1 \vec{v}_0 + \vec{v}_0 \times \vec{v}_1 \end{pmatrix}}$$

Quaternions vs. Matrices

- **Performance (speed)**:
- Rotating a vector:

Rotation matrices ($R\vec{v}$): 15

Quaternions ($\hat{q}\vec{v}\hat{q}^*$): ***30*** or ***41***

$$R\vec{v} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$\boxed{\vec{v}' = \vec{v} + 2\vec{r} \times (\vec{r} \times \vec{v} + w\vec{v})}$$

$$\boxed{\vec{v}' = \hat{q}\vec{v}\hat{q}^*}$$

Quaternions vs. Matrices

- **Performance (general)**:
- Gimbal lock:
- Only a problem with *Euler angles*
- Since one quaternion maps to one rotation…
- No more Euler angles…
- … *no more gimbal lock*!!!

almost rip Apollo 13

Gimbal lock explained:
https://www.youtube.com/watch?v=zc8b2Jo7mno
https://www.youtube.com/watch?v=OmCzZ-D8Wdk

Quaternions vs. Matrices

- **Precision (correctness)**:
- Animation algorithms:
- Cannot animate rotation matrices ☹
- They are also slower to concatenate ☹☹
- Much more efficient to use something we *can* animate (using SLERP)
- Fear not quaternions!!! They are awesome!

Quaternion SLERP vs. Matrix LERP visualized:
https://www.youtube.com/watch?v=uNHIPVOnt-Y

Quaternions vs. Matrices

- **Performance vs. Precision**:
- The ubiquitous computer science dilemma
- Matrices suck but they are used for things quaternions can't handle
- Quaternion ***SLERP*** can be replaced with ***NLERP*** to save some time
- Later we'll discuss how to drop *homogeneous transforms* for a quaternion-related topic ;)

Quaternions in Animation

- **Conversion to rotation matrix**:
- Why would we want to convert a rotation quaternion to a matrix???
- Quaternions are a good *animation tool*…
- …but they do not play nicely with the other children ☹
- GPU does not know quaternions; so we will eventually require matrices for rendering

Quaternions in Animation

- **Conversion to rotation matrix**:
- For the rotation quaternion
  $q = (w, \vec{v}) = (w, x, y, z)$,
  the corresponding rotation matrix is

$$\boldsymbol{R_q} = \begin{bmatrix} \boldsymbol{w^2 + x^2 - y^2 - z^2} & \boldsymbol{2(xy - wz)} & \boldsymbol{2(xz + wy)} \\ \boldsymbol{2(xy + wz)} & \boldsymbol{w^2 - x^2 + y^2 - z^2} & \boldsymbol{2(yz - wx)} \\ \boldsymbol{2(xz - wy)} & \boldsymbol{2(yz + wx)} & \boldsymbol{w^2 - x^2 - y^2 + z^2} \end{bmatrix}$$

Quaternions in Animation

- Pro tip: who says quaternions only represent rotations? ;)

$$\hat{q} = (w, x, y, z)$$ ← *Normalized* quaternions are rotations!

$$s\hat{q} = (sw, sx, sy, sz)$$ ← …what about *un-normalized* quaternions?

Expand this and see what you get… what does the result imply???

$$R_q = \begin{bmatrix} (sw)^2 + (sx)^2 - (sy)^2 - (sz)^2 & 2(sxsy - swsz) & 2(sxsz + swsy) \\ 2(sxsy + swsz) & (sw)^2 - (sx)^2 + (sy)^2 - (sz)^2 & 2(sysz - swsx) \\ 2(sxsz - swsy) & 2(sysz + swsx) & (sw)^2 - (sx)^2 - (sy)^2 + (sz)^2 \end{bmatrix}$$

Quaternions in Animation

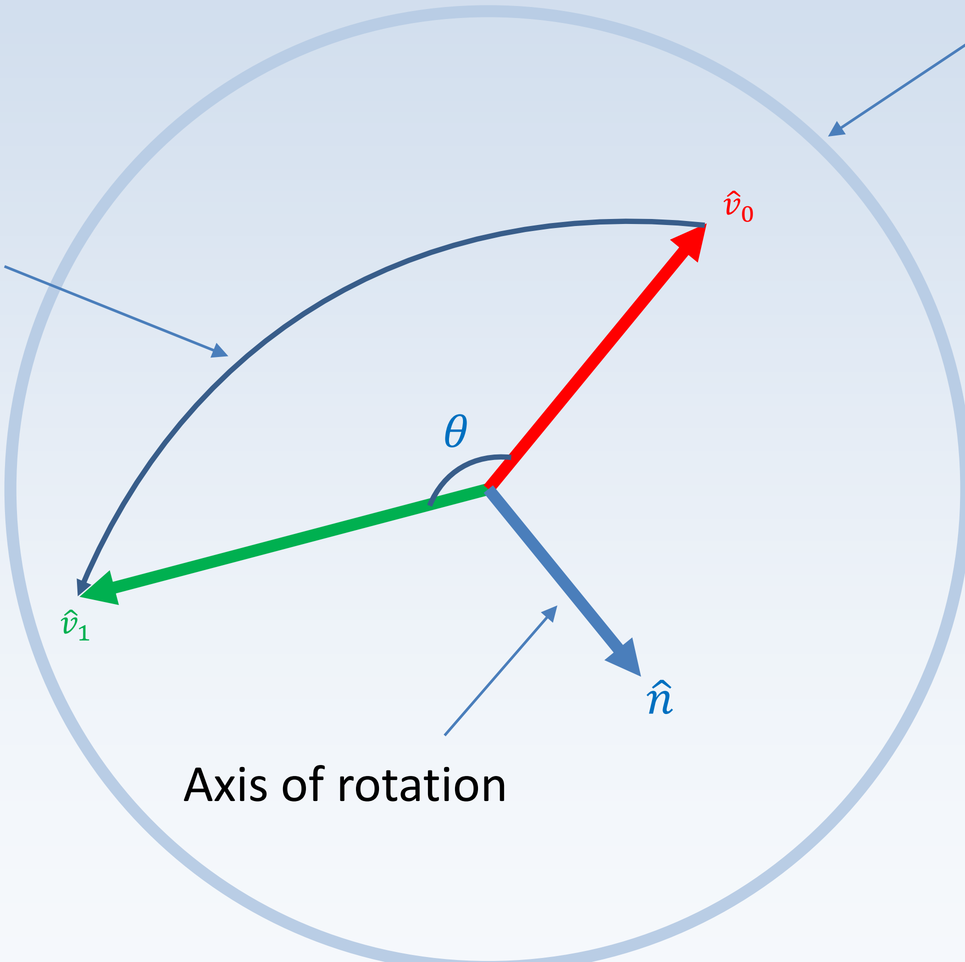
- ***Arcball***:
  https://www.talisman.org/~erlkonig/misc/shoemake92-arcball.pdf
- Concept by *Ken Shoemake* (1992)
  - (this dude is big in quaternion research)
- Project screen coordinates (e.g. from mouse click) on to a sphere
- The “delta” between the two projected screen coordinates is a quaternion!!!

Quaternions in Animation

- ***Arcball***:

Sphere
(the "arcball")

Arc along the sphere between *projected vectors* $\hat{v}_0$ and $\hat{v}_1$



Pro tips:

$$\theta = \cos^{-1}(\hat{v}_0 \cdot \hat{v}_1)$$

$$\hat{n} = \mathrm{normalize}(\hat{v}_0 \times \hat{v}_1)$$

$$\cos\theta = \hat{v}_0 \cdot \hat{v}_1$$

$$\hat{n}\sin\theta = \hat{v}_0 \times \hat{v}_1$$

The end.

- Questions? Comments? Concerns?