

Week 11

3D Modelling	▼
CCSS Technique	▼
Animation and Gimbal Lock	▼
Unit Quaternion	▼
Rotations with Quaternions	▼
Summary	

3D Modelling

Wednesday, November 20, 2019 3:19 PM

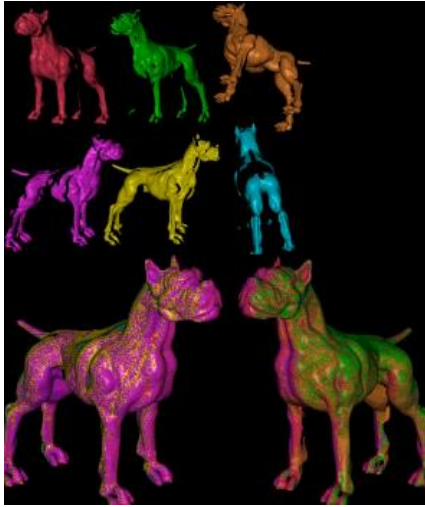
So far we've seen how to draw 3D models while mostly ignoring where they come from

But models need to come from somewhere.

3D Models can come from **three main sources**:

1. Scanning real objects
2. Non-rigid deformation of real object scans (give different shapes)
3. Making synthetic models in 3D modelling softwares (CAD)

Scanning Real Objects



Many 3D scanners are available in the market. Price depends on the resolution of the scan. Examples are:

- Minolta Vivid laser scanner & 3dMD face scanner (expensive)
- Microsoft Kinect (\$200), PrimeSense, Realsense etc.

To cover complete 360 degrees, we must scan the object from multiple directions and then stitch them together

Real scans need a lot of post-processing to remove noise, spikes and cover holes

Non-Rigid Deformation of Linear Models

Deformable models are made by **aligning 3D models** of many real objects

By changing the parameters of the deformable model, we get different 3D shapes that are linear combinations of the original objects

You have already seen how this works in MakeHuman

- MakeHuman was used in the previous project to vary the body shape, skin colour, and facial shape
- MakeHuman also provides a small selection of clothing and a skeleton

Computer Generated Models

Scanning real objects and making deformable models are outside the scope of this unit

We will focus only on how to generate 3D models using computer software

- **MakeHuman** - Makes synthetic humans
- **Blender**
 - Blender includes many different tools useful for different kinds of modelling.
 - We'll focus only on a couple of fundamental techniques: subdivision surfaces and animation via "skinning".

3D Modelling Difficulty

3D modelling can be tedious and time consuming.

- Even positioning a single point in 3D is tricky — Mice and displays are 2D devices
- OpenGL (and DirectX) is based mostly on drawing many triangles.
- So objects must be constructed from many vertices, edges and faces,
- Placing each vertex/edge/face individually is not usually feasible!
- How can we do this quickly and easily?

Modelling Natural Shapes

We can quickly model "blocky" objects — with only a few faces.
But most natural shapes aren't blocky.

We can use prebuilt common shapes like spheres, cylinders, elipsoids.
But these still don't allow us to create "natural" shapes — most shapes in the real world aren't perfect spheres, etc.

Can we generate shapes with many vertices by controlling just a few? We cannot place thousands of vertices, but we can move a few around

Subdivision Surface Method



Subdivision surface method is a method for producing **smooth surfaces** that can be adjusted easily.

The idea is to start with a blocky surface with a manageable number of faces and calculate a smooth surface that roughly follows it.

The smoothing process needs to be predictable.

It is related to earlier techniques, like **NURBS (Non-Uniform Rational B-Splines)** which also use a small number of control points.

- Subdivision surfaces is better for 3D modelling because it doesn't have as strict requirements, such as the points forming a grid of quadrilaterals.
- It is also useful to be able to edit the mesh at the different levels of subdivision, which isn't possible with NURBS and similar techniques.

Catmull-Clark Subdivision Surface Technique is the preferred technique for generating smooth surfaces from a "control mesh" with a relatively small number of points, because it is *simple, predictable and has desirable properties* such as:



- Each original point affects only a small part of the surface
 - We can move different parts of the surface in different directions very easily
 - If you move a point, it affects its neighbor only.
- The 1st derivative is always continuous
 - Its change smoothly and not suddenly
 - A smooth surface is defined as having a derivative that does not change suddenly
 - The normals never change suddenly.
- The 2nd derivative is nearly always continuous
 - The curvature (rate of change of the normals) doesn't change suddenly.
 - The exception is at extraordinary vertices — where the mesh is "irregular", (not a grid of quadrilaterals) (Marked blue in image)

Working with Subdivision Surfaces

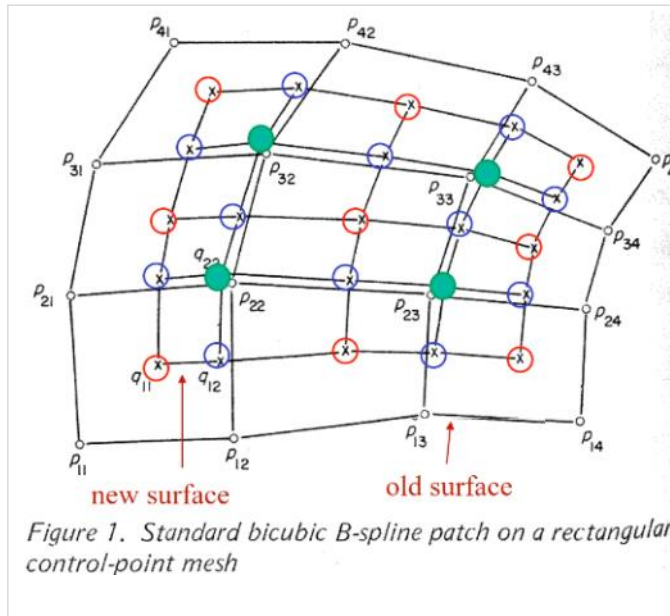
Meshes should usually have mostly quadrilaterals

- Having mostly a grid around the mesh makes it easy to adjust.
- It also tends to smooth well.
- One way to create meshes like these is by "extruding a cube"

To keep things manageable, we work with relatively few control points.

- Just enough to accurately create the desired smooth shape.
- **Loop subdivisions** are often an easy way to add a few points when needed.
- To add many new vertices, subdivide the whole mesh one level.
- You can also select edges to subdivide, although this can cause irregularity. It's better to do whole areas or loops at once.
- Apply the smoothing subdivision just before exporting, or before adding an armature for skinning animation.
- To easily swap between different levels of subdivision, use the multiresolution modifier, which remembers fine detail while editing earlier subdivision levels.
- For 3D detail, try the **sculpt** tool.

Subdivision Step 1



o = old vertices (p_{ij})
 x = new vertices (q_{ij})

After one subdivision step, there is a new vertex created for any:

- Old face (red circle) (In this case: 9)
- Old edge (blue circle) (In this case: 12)
- Old vertex (green circle) (In this case: 4)
 - Outer vertices are not considered as they are at the end of the surface

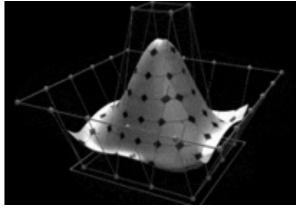
The number of vertices increases greatly with each subdivision (In this case, we go from 4 to 25 ($9+12+4$))

Subdivision Step 2

We refer to the new vertices as points and the old vertices as vertices

<p>New "face" points are at the average of the vertices for the face (e.g. average of 4 points)</p> <p>New "edge" points are at the average of the two vertices on the edge and the two face points on either side of the edge</p>	<p>The placement of new "vertex" points:</p> <p>For the vertex P, a new point is placed at:</p> $\frac{F + 2E + (n - 3)P}{n}$ <p>F = Average of the face points (e.g. 4 points) E = Average of the edge points (e.g. 4 points) n = Number of edges that connect to P (e.g. 4 edges)</p> <p>The faces and edges are the original ones that touch the original P</p>
--	---

Important Properties



When the control points form a simple grid topology (open surface), the surface tends towards a bicubic B-Spline - a standard kind of surface used when smoothness is required.

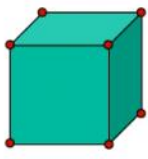
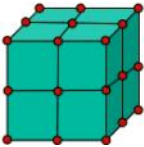
Unlike other techniques for generating such surfaces (like computationally intensive NURBS), *the technique naturally extends to other topologies, giving 3D modelers much freedom.*

Properties like texture coordinates can be smoothly generated in the same we previously generated the vertex positions - by averaging them with the same weights during subdivision.

Closed Surfaces

Counting the number of new vertices for open surfaces after one subdivision step can be a bit confusing. For closed surfaces, the counting is easier and more intuitive.

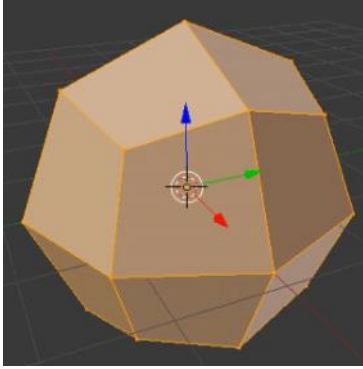
Example: A Cube

	<p>For a cube, initially</p> <p>There are 8 vertices, 6 faces, and 12 edges.</p> $V_0 = 8$ $F_0 = 6$ $E_0 = 12$	<p>In General, after n subdivision steps:</p> $V_n = V_{n-1} + F_{n-1} + E_{n-1}$ $F_n = 4F_{n-1}$ $E_n = 2E_{n-1} + 4F_{n-1}$
<p>- After one subdivision step, how many vertices are there?</p> 	<p>There are 26 vertices, 24 faces, and 48 edges.</p> $V_1 = V_0 + F_0 + E_0 = 26$ $F_1 = 4F_0 = 24$ $E_1 = 2E_0 + 4F_0 = 24 + 24 = 48$	<p>Thus, after two subdivision steps:</p> $V_2 = V_1 + F_1 + E_1 = 26 + 24 + 48 = 98 \text{ vertices.}$

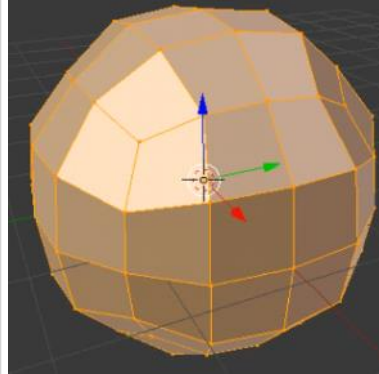
As the Catmull and Clark subdivision surface method constrains the surface to be smooth, the cube would **approach the shape of a sphere** after a few subdivisions.

Examples in Blender

A cube after 1 subdivision step



A cube after 2 subdivision steps



Animation and Gimbal Lock

Wednesday, November 20, 2019 3:19 PM

Static Objects

Look at your project scene objects

- They just sit there - they are rigid objects
- We can move them to other places and alter the rotation and colour, but they're still boring

What about more complicated & interesting things?

- Living things
- Vehicles
- Things that perform movement

These things are all made up of separate moving parts that are still part of the whole object

- Where I go, my arms go with me
- Its parts move with the main body

Role of An Artist

Create a mesh for a game shape/character/vehicle

- Geometry
- Texturing
- Material properties

Create animation sequences for different game states for that entity

Note: Exporting data from modelling/animation tools so that it can be imported into another application can be tricky.

Keyframe/Cell Animation

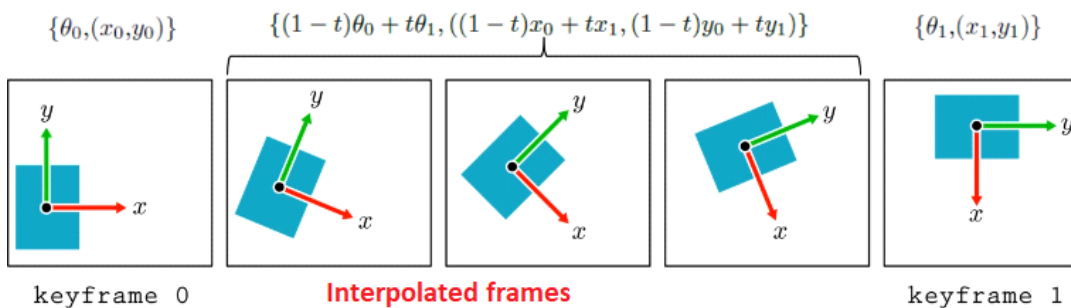
In traditional hand-drawn animation:

- The senior key artist would draw the **keyframes**
- The junior artist would fill the **in-between frames**

We can use a similar approach for computer animation

For a 30-fps animation, less than 30 keyframes need to be defined per second:

1. The data (e.g., joint angles of an articulated figure) are assigned to the keyframes
2. In-between frames are interpolated



Tank Example

Gun Turret should be able to rotate

Gun Barrel should be able to move up and down

The wheels should be able to spin

Note: keyframes do not need to be equi-distant apart.



Keyframe 1

Set the initial position of the gun turret



Keyframe 2

Set the new desired position of the gun turret



Keyframe 3

Set the new desired position of the gun barrel

Tank Hierarchy

Tank Hull — this is the central parent section

- Wheels — these are children of the hull and are siblings to each other
- Gun Turret — this is another child of the hull
 - Gun Barrel — this is the only child of the Gun Turret

The animation must take into account the hierarchy of parts:

- Where the hull goes, the wheels go
- When the hull rotates, the wheels rotate

Note: The hull, wheels, etc., are represented as 3D meshes (i.e., triangles), so when the hull moves, it is actually the vertices of the triangles being transformed.

Interpolating Euler Angles

The Euler angles (pitch, yaw, and roll) are often used (e.g. in blender) to represent the rotation angles of a body limb about a body joint, e.g.,

- In keyframe 1 (frame number 11): we have a_1 , B_1 and y_1 and representing the pitch, yaw, and roll angles of a rotation.
- In keyframe 2 (frame number 21): we have a_1 , B_1 and y_1 and representing the pitch, yaw, and roll angles of another rotation.
- **We need to compute interpolated Euler angle for frames 12 to 20.**

Unfortunately, the Euler angles are problematic when interpolating:

- Unexpected rotations would result when we combine the three interpolated angles, e.g. consider interpolating between $a_1 = -170$ and $a_2 = 170$.
- A **Gimbal Lock** can occur

Gimbal Lock

Gimbal Lock refers to the situation where a rotation accidentally causes two local coordinate axes to align and thus 1 degree of freedom is lost (We are *left with two effective rotation axes rather than three*).

This is a serious problem as using the Euler angles to represent rotations that **we are not able to rotate about one of the 3 principal axes** (x, y, and z) when Gimbal Lock occurs.

[YouTube video](#)

Avoiding Gimbal Lock

Can we avoid gimbal lock in interpolating rotations of keyframes if we want to remain using the Euler angles?

Consider the three Euler angles: pitch (θ_x), yaw (θ_y), and roll (θ_z) angles. We can use any of the 6 multiplication orders

1. $R_x(\theta_x)R_y(\theta_y)R_z(\theta_z)$
2. $R_x(\theta_x)R_z(\theta_z)R_y(\theta_y)$
3. $R_y(\theta_y)R_x(\theta_x)R_z(\theta_z)$
4. $R_y(\theta_y)R_z(\theta_z)R_x(\theta_x)$
5. $R_z(\theta_z)R_x(\theta_x)R_y(\theta_y)$
6. $R_z(\theta_z)R_y(\theta_y)R_x(\theta_x)$

As the multiplication of matrices is not commutative, the resultant rotation matrix is different for each order. You will need to work out different Euler angles for each order, so that the resultant rotation matrix is the desired one.

If using multiplication order #6 leads to a gimbal lock problem, say, then we can use another multiplication order. **Note that this does not eliminate gimbal lock - we only avoid the problem for a particular case/interpolation.**

Unit Quaternions offer an elegant solution to the Gimbal Lock problem

$a + bi + cj + dk$ where $a, b, c, d \in \mathbb{R}$, and i, j, k satisfy $i^2 = j^2 = k^2 = -1$	Quaternions are an extension of complex numbers Consist of real numbers (a, b ...) and complex ones (i,j,k)
4D-vectors $\mathbf{q} = (a, b, c, d)$ For compactness, we write $\mathbf{q} = (a, \mathbf{v})$, where $a \in \mathbb{R}$ and $\mathbf{v} \in \mathbb{R}^3$	An alternative representation (more suitable for rotations) considers quaternions as 4D vectors In compact, a is a scalar, v is a vector
$a = 0$	Pure quaternion condition
$\ \mathbf{q}\ = 1$	Unit quaternion condition <u>Rotations can be represented using unit quaternions.</u>

Quaternion Advantages

- Interpolation is easy between 2 quaternions.
- Smoother animation can be achieved.
- Compared to matrices, quaternions take up less storage space. (e.g. 4 numbers versus $4 \times 4 = 16$ numbers)
- Quaternions can be easily converted to matrices for rendering.
- Quaternions do not suffer from gimbal lock.

Angle-Axis Rotations

We use **unit quaternions** to represent **3D rotations**

if we use $\mathbf{q} = (a, \mathbf{v})$ to represent a rotation then we normalize \mathbf{q} so that $\|\mathbf{q}\| = 1$.

Any 3D rotation about the origin can be described by defining:

- A new rotation axis \mathbf{w}
 - $\mathbf{w} = (w_1, w_2, w_3)$ (assuming that $\|\mathbf{w}\| = 1$) in the 3D space
- A rotation angle θ about that axis

The corresponding unit quaternion is:

$$\left(\cos\left(\frac{\theta}{2}\right), \mathbf{w} \sin\left(\frac{\theta}{2}\right) \right)$$

Examples of Unit Quaternions

First part is always $\cos(\text{angle}/2)$

Second has $\sin(\text{angle}/2)$ in spot of axis

$(\cos(\frac{\alpha}{2}), \sin(\frac{\alpha}{2}), 0, 0)$	X axis rotation
$(\cos(\frac{\beta}{2}), 0, \sin(\frac{\beta}{2}), 0)$	Y axis rotation
$(\cos(\frac{\gamma}{2}), 0, 0, \sin(\frac{\gamma}{2}))$	Z axis rotation

Multi-axis rotations will require more complicated quaternions

Inverse Unit Quaternions

We know that the inverse of a rotation matrix \mathbf{R} is \mathbf{R}^T .

Let $\mathbf{q} = (a, \mathbf{v})$ be the quaternion representing \mathbf{R} , then $\mathbf{q}^{-1} = (a, -\mathbf{v})$ is the inverse of \mathbf{q} representing \mathbf{R}^T .	We simply inverse the vector's sign
--	-------------------------------------

The quaternion $(\mathbf{1}, \mathbf{0}, \mathbf{0}, \mathbf{0})$ represents the **identity matrix** (zero rotation angle).

Quaternion Multiplication

Multiplying two rotation matrices together is equivalent to:

$\mathbf{q}_1 = (s_1, \mathbf{v}_1)$ $\mathbf{q}_2 = (s_2, \mathbf{v}_2)$	Scalar and vector values
--	--------------------------

$$\mathbf{q}_1 \mathbf{q}_2 = (s_1 s_2 - \mathbf{v}_1 \cdot \mathbf{v}_2, s_1 \mathbf{v}_2 + s_2 \mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2)$$

where \mathbf{x} is cross product, and \bullet is dot product.

Similar to matrix multiplication, quaternion multiplication is not commutative.

Verifying

An easy example that we can verify: Let rotation R_1 be the rotation $\text{rotation}_x(\alpha)$ and R_2 be $\text{rotation}_x(\beta)$. Then we know the resultant rotation R should be $\text{rotation}_x(\alpha + \beta)$. The corresponding unit quaternions are: $\mathbf{q}_1 = (\cos(\frac{\alpha}{2}), \sin(\frac{\alpha}{2}), 0, 0)$ and $\mathbf{q}_2 = (\cos(\frac{\beta}{2}), \sin(\frac{\beta}{2}), 0, 0)$. Applying the multiplication formula

$$\mathbf{q} = \begin{pmatrix} \cos(\alpha/2)\cos(\beta/2) - \sin(\alpha/2)\sin(\beta/2) \\ \cos(\alpha/2)\sin(\beta/2) + \cos(\beta/2)\sin(\alpha/2) \\ 0 \\ 0 \end{pmatrix}$$

Recall that $\mathbf{q}_1 \mathbf{q}_2 = (s_1 s_2 - \mathbf{v}_1 \cdot \mathbf{v}_2, s_1 \mathbf{v}_2 + s_2 \mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2)$. Here we have $s_1 = \cos(\alpha/2)$, $s_2 = \cos(\beta/2)$, $\mathbf{v}_1 = (\sin(\alpha/2), 0, 0)$, and $\mathbf{v}_2 = (\sin(\beta/2), 0, 0)$.

Using the rules

$\cos(A + B) = \cos(A) \cos(B) - \sin(A) \sin(B)$ and

$\sin(A + B) = \sin(A) \cos(B) + \cos(A) \sin(B)$

we get the resultant quaternion

$$\mathbf{q} = \begin{pmatrix} \cos(\frac{\alpha+\beta}{2}) \\ \sin(\frac{\alpha+\beta}{2}) \\ 0 \\ 0 \end{pmatrix}$$

which represents the rotation about the x-axis an angle $\alpha + \beta$.

Rotations with Quaternions

Wednesday, November 20, 2019 3:19 PM

Rotating Points and Vectors

Let $\mathbf{p} = (x, y, z)$ and $\mathbf{v} = (u, v, w)$ be a point and a vector in 3D. Let $\mathbf{q} = (a, \mathbf{w})$ be the unit quaternion representing the 3D rotation matrix \mathbf{R} . Suppose that we want to transform \mathbf{p} and \mathbf{v} by \mathbf{R} . i.e., we want to compute $\mathbf{p}' = \mathbf{R}\mathbf{p}$ and $\mathbf{v}' = \mathbf{R}\mathbf{v}$. How do we achieve the same result using the unit quaternion \mathbf{q} ?

We need to rewrite both \mathbf{p} and \mathbf{q} as 4D entities by letting $\mathbf{p} = (0, x, y, z)$ and $\mathbf{v} = (0, u, v, w)$. Then

$$\begin{aligned}\mathbf{p}' &= \mathbf{q} \mathbf{p} \mathbf{q}^{-1} \\ \mathbf{v}' &= \mathbf{q} \mathbf{v} \mathbf{q}^{-1}\end{aligned}$$

where \mathbf{q}^{-1} denotes the inverse of \mathbf{q} .

To create 4D entities, We append a 0 or 1 to front. Since unit quaternions cannot be used for scaling and translation it doesn't matter which is chosen.

Then we pre multiply with \mathbf{q} and post multiply with \mathbf{q} inverse.

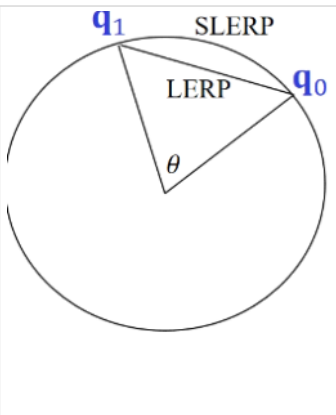
Both the outputs \mathbf{p}' and \mathbf{v}' are 4D entities can be converted back to the 3D space by simply dropping their 1st component.

Interpolating Quaternions

Interpolation is needed in animation

There are two main methods of interpolating quaternions: **LERP** and **SLERP**

The functions give an interpolated quaternion for a given t value

	<p>– LERP (Linear interpolation)</p> $\text{LERP}(\mathbf{q}_0, \mathbf{q}_1, t) = (1 - t)\mathbf{q}_0 + t\mathbf{q}_1$	<p>We move from Q0 to Q1 in linear steps</p>
	<p>– SLERP (spherical linear interpolation)</p> $\text{SLERP}(\mathbf{q}_0, \mathbf{q}_1, t) = \frac{\sin((1 - t)\theta)\mathbf{q}_0}{\sin(\theta)} + \frac{\sin(t\theta)\mathbf{q}_1}{\sin(\theta)}$ <p>where \mathbf{q}_0 and \mathbf{q}_1 are the quaternions that describe the initial and final orientations (= rotation), parameter $t = 0 \dots 1$, and θ is the angle between \mathbf{q}_0 and \mathbf{q}_1, i.e.,</p> $\theta = \arccos(\mathbf{q}_0 \cdot \mathbf{q}_1)$	<p>We move on a circle/sphere.</p> <p>The intermediate points are given by the equation</p>

SLERP

$$\text{SLERP}(\mathbf{q}_0, \mathbf{q}_1, t) = \frac{\sin((1-t)\theta)}{\sin(\theta)} \mathbf{q}_0 + \frac{\sin(t\theta)}{\sin(\theta)} \mathbf{q}_1$$

← weights →

Note that SLERP is usually implemented differently in practice. Also, **normalization would be required** since the two weights do not sum to 1 as in LERP.

What we aim to achieve is the shortest trajectory (the *geodesic*) that brings the initial orientation \mathbf{q}_0 of the object to its final orientation \mathbf{q}_1 . Mathematically, we find $\Delta\mathbf{q}$ such that $\mathbf{q}_1 = \Delta\mathbf{q} \mathbf{q}_0$, i.e.,

$$\Delta\mathbf{q} = \mathbf{q}_1 \mathbf{q}_0^{-1}$$

then at each $t \in [0,1]$, the interpolated quaternion is computed as $(\Delta\mathbf{q})^t \mathbf{q}_0$.

SLERP gives smoother intermediate steps but requires more computations.

4

Considerations

Unit quaternion is **not a unique representation** for rotations. One rotation can have 2 representations

Both $\mathbf{q} = (a, \mathbf{v})$ and $\mathbf{q}' = (-a, -\mathbf{v})$ represent the same rotation. If one corresponds to a 160° rotation then the other corresponds to a 200° in the other direction.

To avoid interpolating along the longer path, change the sign of the quaternion so that its **1st component** (representing $\cos(\theta/2)$) is always **positive**.

This will ensure that the rotation angle θ is always less than 180° .

Converting UQs to Rotation Matrices

The easiest way to convert a unit quaternion $\mathbf{q} = (a, \mathbf{v})$ to a 3×3 rotation matrix \mathbf{R} is using the **Rodrigues' rotation formula**:

$$\begin{aligned}\mathbf{R} &= \mathbf{I} + \sin(\theta) \text{skew}(\mathbf{v}) + (1 - \cos(\theta)) \mathbf{v} \mathbf{v}^T \\ \mathbf{R}^T &= \mathbf{I} - \sin(\theta) \text{skew}(\mathbf{v}) + (1 - \cos(\theta)) \mathbf{v} \mathbf{v}^T\end{aligned}$$

where

- \mathbf{I} denotes the 3×3 identity matrix;
- θ can be obtained from the 1st component of \mathbf{q} as follows: $\theta = 2 \arccos(a)$;
- $\text{skew}(\mathbf{v})$ denotes the skew-symmetric matrix composed from the elements of $\mathbf{v} = (v_1, v_2, v_3)$:

$$\text{skew}(\mathbf{v}) = \begin{bmatrix} 0 & -v_3 & v_2 \\ v_3 & 0 & -v_1 \\ -v_2 & v_1 & 0 \end{bmatrix}$$

Quaternions in OpenGL

OpenGL doesn't support them directly, but implementing quaternions is not difficult.

To do so:

1. Do all the rotation interpolation between frames using unit quaternions.
2. Convert the interpolated results to matrices and combine them with the necessary translation and scaling matrices.
3. Pass the resultant matrices to the vertex shader.

Summary

In summary, to generate smooth animation, we do the following:

- Define the pose (orientation/rotation and position) of each object in keyframes.
- Option 1 (preferred): Try to use quaternions to represent rotations to avoid gimbal lock. Interpolate quaternions to get the rotations for the in-between frames. Get the resultant rotation matrices for these frames.
- Option 2: Try to avoid gimbal lock by change the order of applying the roll, pitch, and yaw in the rotation. Will need different orders for different parts of the animation.
- Combine the rotation with scaling and translation etc. to get the resultant transformation matrices for the vertex shader.