Session 3: Motion Tracking Fundamentals - Study Notes

Table of Contents

1. Why This Matters for OptiTrack/VR
2. 3D Coordinate Systems
3. Vectors - The Foundation
4. Vector Operations
5. Quaternions - The Right Way to Rotate
6. Transforms - Position + Rotation
7. Real-World Applications
8. Common Interview Questions

1. Why This Matters for OptiTrack/VR

What OptiTrack Does

OptiTrack cameras track **reflective markers** in 3D space. When you wear a VR headset with markers attached:

1. Cameras detect marker positions (x, y, z coordinates)
2. Software calculates the headset's **position** and **rotation**
3. VR system uses this to update what you see

The Math You Need

- **Vectors:** Represent positions, directions, velocities
- **Quaternions:** Represent rotations (better than Euler angles)
- **Transforms:** Combine position + rotation to describe object state

In interviews, they want to see you understand the 3D math behind tracking systems.

2. 3D Coordinate Systems

Right-Handed Coordinate System (OptiTrack Standard)

```
       Y (up)
       |
       |
       +------ X (right)
      /
     /
    Z (forward toward you)
```

Conventions:

- **X-axis:** Right
- **Y-axis:** Up
- **Z-axis:** Forward (toward you in OpenGL/OptiTrack)

World Space vs Local Space

World Space: The global coordinate system (the room)

```
VR Headset at world position (2, 1.5, 3)
```

Local Space: Relative to an object

```
Marker on headset at local position (0.1, 0.05, -0.1) relative to headset center
```

Transformation: Converting local → world coordinates

```
World Position = Object Position + Rotate(Local Position)
```

3. Vectors - The Foundation

What is a Vector?

A vector is a **direction and magnitude** in 3D space.

```
struct Vec3 {
    float x, y, z;
};

Vec3 position = {1.0f, 2.0f, 3.0f};     // A point in space
Vec3 velocity = {0.5f, 0.0f, -0.2f};    // Movement direction and speed
Vec3 forward = {0.0f, 0.0f, -1.0f};     // Direction headset is facing
```

Visualizing Vectors

```
    (x=2, y=3)
      *
     /|
    / |
   /  | y=3
  /   |
 /____|
   x=2

Magnitude (length) = sqrt(2² + 3²) = sqrt(13) ≈ 3.6
```

Types of Vectors

Position Vector: A point in space

```
Vec3 headsetPos = {1.5f, 1.8f, 0.5f};  // Headset location
```

Direction Vector: Which way something is pointing (usually normalized)

```
Vec3 forward = {0.0f, 0.0f, -1.0f};    // Looking forward
```

Velocity Vector: Speed and direction of movement

```
Vec3 velocity = {0.1f, 0.0f, 0.0f};    // Moving right at 0.1 units/frame
```

4. Vector Operations

Vector Addition (+)

Use case: Applying offsets, combining movements

```
v1 = (1, 2, 3)
v2 = (4, 5, 6)
v1 + v2 = (1+4, 2+5, 3+6) = (5, 7, 9)
```

Example:

```
Vec3 position = {5.0f, 2.0f, 0.0f};
Vec3 offset = {0.1f, 0.0f, 0.0f};
Vec3 newPosition = position + offset;  // Move 0.1 units to the right
```

Formula:

```
Vec3 operator+(const Vec3& other) const {
    return {x + other.x, y + other.y, z + other.z};
}
```

Vector Subtraction (-)

Use case: Finding direction between two points

```
v1 = (5, 3, 2)
v2 = (2, 1, 1)
v1 - v2 = (3, 2, 1)  // Direction from v2 to v1
```

Example:

```
Vec3 headPos = {2.0f, 1.5f, 0.0f};
Vec3 controllerPos = {1.0f, 1.2f, 0.5f};
Vec3 direction = headPos - controllerPos;  // Vector pointing from controller to head
```

Formula:

```
Vec3 operator-(const Vec3& other) const {
    return {x - other.x, y - other.y, z - other.z};
}
```

Scalar Multiplication (*)

Use case: Scaling vectors (change magnitude, keep direction)

```
v = (2, 3, 1)
v * 2 = (4, 6, 2)  // Twice as long, same direction
```

Example:

```
Vec3 velocity = {1.0f, 0.0f, 0.0f};
Vec3 fastVelocity = velocity * 3.0f;  // 3x speed
```

Formula:

```
Vec3 operator*(float scalar) const {
    return {x * scalar, y * scalar, z * scalar};
}
```

Magnitude (Length)

Use case: How far? How fast? Distance from origin.

```
v = (3, 4, 0)
|v| = sqrt(3² + 4² + 0²) = sqrt(9 + 16) = sqrt(25) = 5
```

Example:

```
Vec3 velocity = {3.0f, 4.0f, 0.0f};
float speed = velocity.magnitude();  // = 5.0
```

Formula:

```
float magnitude() const {
    return sqrt(x*x + y*y + z*z);
}
```

Normalization (Unit Vector)

Use case: Direction without magnitude (length = 1)

```
v = (3, 4, 0)
|v| = 5
normalized = (3/5, 4/5, 0/5) = (0.6, 0.8, 0)
|normalized| = 1
```

Example:

```
Vec3 direction = {3.0f, 4.0f, 0.0f};
Vec3 unitDirection = direction.normalize();  // (0.6, 0.8, 0) - same direction, length
```

Formula:

```
Vec3 normalize() const {
    float mag = magnitude();
    if (mag == 0) return {0, 0, 0};  // Avoid division by zero
    return {x / mag, y / mag, z / mag};
}
```

Why normalize?

- Rotations require unit vectors
- Comparing directions (ignoring magnitude)
- Consistent movement speed

Dot Product (·)

Use case: Angle between vectors, projection, checking if perpendicular

```
v1 = (1, 0, 0)
v2 = (0, 1, 0)
v1 · v2 = (1×0) + (0×1) + (0×0) = 0  (perpendicular!)
```

Formula:

```
float dot(const Vec3& other) const {
    return x*other.x + y*other.y + z*other.z;
}
```

Relationship to angle:

```
v1 · v2 = |v1| × |v2| × cos(θ)

For unit vectors:
v1 · v2 = cos(θ)
```

Interpretation:

- `dot = 1` : Same direction (0° angle)
- `dot = 0` : Perpendicular (90° angle)
- `dot = -1` : Opposite direction (180° angle)
- `dot > 0` : Less than 90° apart
- `dot < 0` : More than 90° apart

Example:

```
Vec3 forward = {0, 0, -1};
Vec3 toTarget = {1, 0, -1};
float alignment = forward.normalize().dot(toTarget.normalize());
// If alignment > 0.7, target is in front of you
```

Cross Product (×)

Use case: Find perpendicular vector, calculate rotation axis

```
v1 = (1, 0, 0)  // X-axis
v2 = (0, 1, 0)  // Y-axis
v1 × v2 = (0, 0, 1)  // Z-axis (perpendicular to both)
```

Formula:

```
Vec3 cross(const Vec3& other) const {
    return {
        y * other.z - z * other.y,  // x component
        z * other.x - x * other.z,  // y component
        x * other.y - y * other.x   // z component
    };
}
```

Properties:

- **Order matters:** `v1 × v2 = -(v2 × v1)`
- **Right-hand rule:** Curl fingers from v1 to v2, thumb points to result
- **Result is perpendicular** to both input vectors
- **Magnitude = |v1| × |v2| × sin(θ)** (area of parallelogram)

Example:

```
Vec3 up = {0, 1, 0};
Vec3 forward = {0, 0, -1};
Vec3 right = up.cross(forward);  // (1, 0, 0)
```

5. Quaternions - The Right Way to Rotate

The Problem with Euler Angles

Euler angles: Rotation as three angles (pitch, yaw, roll)

```
struct EulerAngles {
    float pitch;  // Rotation around X (nodding)
    float yaw;    // Rotation around Y (shaking head "no")
    float roll;   // Rotation around Z (tilting head)
};
```

Problems:

1. **Gimbal Lock:** At certain angles, you lose a degree of freedom
    - Pitch to 90° → roll and yaw become the same axis!
2. **Interpolation:** Can't smoothly blend between rotations
3. **Order dependent:** XYZ rotation ≠ ZYX rotation

Why VR cares: When you look up (pitch 90°), the headset can't distinguish roll from yaw = BAD tracking!

What is a Quaternion?

A quaternion is a **4D number** that represents rotation:

```
struct Quaternion {
    float x, y, z;  // Vector part (rotation axis)
    float w;        // Scalar part (rotation amount)
};
```

Think of it as: "Rotate by angle θ around axis (x, y, z)"

Axis-Angle Representation

Most intuitive way to think about rotation:

- **Axis:** Direction vector (normalized)
- **Angle:** How much to rotate around that axis

Example:

```
Rotate 90° around Y-axis:
  axis = (0, 1, 0)
  angle = π/2 (90 degrees in radians)
```

Converting to Quaternion:

```
Quaternion(Vec3 axis, float angleRadians) {
    Vec3 normalizedAxis = axis.normalize();
    float halfAngle = angleRadians / 2.0f;
    float sinHalf = sin(halfAngle);

    x = normalizedAxis.x * sinHalf;
    y = normalizedAxis.y * sinHalf;
    z = normalizedAxis.z * sinHalf;
    w = cos(halfAngle);
}
```

Why half angle? Math reasons (quaternion double-cover property) - just remember to use `angle/2` .

Quaternion Multiplication (Combining Rotations)

Use case: Apply multiple rotations

```
Quaternion rotate90Y(Vec3{0,1,0}, M_PI/2);   // Rotate 90° around Y
Quaternion rotate45Z(Vec3{0,0,1}, M_PI/4);   // Then rotate 45° around Z
Quaternion combined = rotate45Z * rotate90Y; // Combined rotation
```

Important: Order matters! `q1 * q2 ≠ q2 * q1`

Formula: (You don't need to memorize, just implement from TODO)

```
Quaternion operator*(const Quaternion& q) const {
    Quaternion result;
    result.w = w*q.w - x*q.x - y*q.y - z*q.z;
    result.x = w*q.x + x*q.w + y*q.z - z*q.y;
    result.y = w*q.y - x*q.z + y*q.w + z*q.x;
    result.z = w*q.z + x*q.y - y*q.x + z*q.w;
    return result;
}
```

Rotating a Vector with a Quaternion

Use case: Apply headset rotation to a forward vector

```
Vec3 localForward = {0, 0, -1};
Quaternion headsetRotation = ...;
Vec3 worldForward = headsetRotation.rotate(localForward);
```

Simplified formula: (Efficient version)

```
Vec3 rotate(const Vec3& v) const {
    Vec3 qvec = {x, y, z};
    Vec3 uv = qvec.cross(v);
    Vec3 uuv = qvec.cross(uv);
    return v + (uv * (2.0f * w)) + (uuv * 2.0f);
}
```

Identity Quaternion (No Rotation)

```
Quaternion identity = {0, 0, 0, 1};  // No rotation
```

Why Quaternions are Better

Feature	Euler Angles	Quaternions
Gimbal lock	✗ Yes	✓ No
Smooth interpolation	✗ Hard	✓ Easy (SLERP)
Combining rotations	✗ Complex	✓ Simple (multiply)
Memory	3 floats	4 floats
Human-readable	✓ Yes	✗ No

For VR/OptiTrack: Always use quaternions for internal representation.

6. Transforms - Position + Rotation

What is a Transform?

A **transform** describes an object's state in 3D space:

```
class Transform {
    Vec3 position;        // Where is it?
    Quaternion rotation;  // Which way is it facing?
};
```

(Sometimes also includes `scale`, but OptiTrack rigid bodies don't scale)

Local to World Transformation

Problem: You have a marker's position relative to a headset. Where is it in the room?

```
Vec3 transformPoint(const Vec3& localPoint) const {
    // 1. Rotate the local point by the object's rotation
    Vec3 rotated = rotation.rotate(localPoint);

    // 2. Add the object's position
    return rotated + position;
}
```

Example:

```
Headset Transform:
  position = (2, 1.5, 3)
  rotation = 90° around Y

Local marker position: (0.1, 0, 0)  // 0.1m to the right of headset center

World position = rotate(0.1, 0, 0) + (2, 1.5, 3)
               = (0, 0, -0.1) + (2, 1.5, 3)    // After 90° Y rotation, right becomes -
               = (2, 1.5, 2.9)
```

Direction Vectors

Objects have **local** direction vectors:

```
Vec3 localForward = {0, 0, -1};  // -Z is forward (OpenGL convention)
Vec3 localUp = {0, 1, 0};        // +Y is up
Vec3 localRight = {1, 0, 0};     // +X is right
```

Getting world-space directions:

```
Vec3 forward() const {
    return rotation.rotate({0, 0, -1});
}

Vec3 up() const {
    return rotation.rotate({0, 1, 0});
}

Vec3 right() const {
    return rotation.rotate({1, 0, 0});
}
```

Use case:

```
Transform headset = ...;
Vec3 lookDirection = headset.forward();  // Which way is the user looking?

// Move 0.5 units in the direction the headset is facing
Vec3 newPos = headset.getPosition() + (lookDirection * 0.5f);
```

7. Real-World Applications

OptiTrack Tracking Pipeline

```
1. Cameras detect marker positions (2D in each camera)
   ↓
2. Triangulation → 3D marker positions in world space
   ↓
3. Marker clustering → Identify which markers belong to which rigid body
   ↓
4. Rigid body pose estimation → Calculate position + rotation (quaternion)
   ↓
5. Send to VR application → Update headset/controller transforms
```

Common Calculations

Distance between headset and controller:

```
Vec3 headPos = headset.getPosition();
Vec3 controllerPos = controller.getPosition();
Vec3 diff = headPos - controllerPos;
float distance = diff.magnitude();
```

Is target in front of headset?

```
Vec3 toTarget = targetPos - headsetPos;
Vec3 forward = headset.forward();
float alignment = toTarget.normalize().dot(forward);
if (alignment > 0.7) {  // cos(45°) ≈ 0.7
    // Target is in front
}
```

Calculate velocity:

```
Vec3 currentPos = headset.getPosition();
Vec3 previousPos = ...;
float deltaTime = 1.0f / 120.0f;  // 120 FPS
Vec3 velocity = (currentPos - previousPos) / deltaTime;
```

Smooth rotation interpolation (SLERP):

```
// Blend between two rotations smoothly
Quaternion slerp(Quaternion q1, Quaternion q2, float t) {
    // t = 0 → q1, t = 1 → q2, t = 0.5 → halfway
    // (Implementation complex, but concept simple)
}
```

8. Common Interview Questions

Conceptual Questions

Q: What's the difference between a position and a direction vector?

- Position: A point in space (origin matters)
- Direction: A direction and magnitude (origin doesn't matter, often normalized)

Q: Why use quaternions instead of Euler angles?

- No gimbal lock
- Smooth interpolation
- Easy to combine rotations
- Better for real-time systems

Q: What does the dot product tell you?

- How aligned two vectors are
- If result > 0: less than 90° apart
- If result = 0: perpendicular
- If result < 0: more than 90° apart

Q: What does the cross product give you?

- A vector perpendicular to both input vectors
- Direction follows right-hand rule
- Used to find rotation axes

Q: How do you convert from local space to world space?

1. Rotate the local point by the object's rotation
2. Add the object's position

Practical Questions

Q: Given two points, how do you find the direction from A to B?

```
Vec3 direction = (B - A).normalize();
```

Q: How do you check if a point is in front of the camera?

```
Vec3 toPoint = (point - cameraPos).normalize();
Vec3 forward = camera.forward();
float dot = toPoint.dot(forward);
if (dot > 0) { /* in front */ }
```

Q: How do you rotate a vector 90° around the Y-axis?

```
Quaternion rot(Vec3{0,1,0}, M_PI/2);
Vec3 rotated = rot.rotate(originalVec);
```

Quick Reference Formulas

```
// Vector Operations
magnitude = sqrt(x² + y² + z²)
normalize = v / magnitude
dot(v1, v2) = v1.x*v2.x + v1.y*v2.y + v1.z*v2.z
cross(v1, v2) = (v1.y*v2.z - v1.z*v2.y,
                 v1.z*v2.x - v1.x*v2.z,
                 v1.x*v2.y - v1.y*v2.x)

// Quaternion from axis-angle
half = angle / 2
q.x = axis.x * sin(half)
q.y = axis.y * sin(half)
q.z = axis.z * sin(half)
q.w = cos(half)

// Transform point
worldPos = rotation.rotate(localPos) + position
```

Key Takeaways

✓ **Vectors** represent positions, directions, velocities in 3D

✓ **Dot product** measures alignment (cos of angle)

✓ **Cross product** finds perpendicular vectors
✓ **Normalize** to get direction without magnitude
✓ **Quaternions** avoid gimbal lock (critical for VR!)
✓ **Transforms** combine position + rotation
✓ **Local → World** transformation: rotate then translate

You're now equipped to tackle Session 3 exercises! 🚀